Microsoft®

5.0

MACRO ASSEMBLER

FOR THE MS-DOS® OPERATING SYSTEM

PROGRAMMER'S GUIDE

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser's personal use without the written permission of Microsoft Corporation.

© Copyright Microsoft Corporation, 1987. All rights reserved.
Simultaneously published in the U.S. and Canada.

Microsoft®, MS®, MS-DOS®, XENIX®, and CodeView® are registered trademarks of
Microsoft Corporation.

IBM® is a registered trademark of International Business Machines Corporation.

Intel® is a registered trademark of Intel Corporation.

ProKey® is a registered trademark of RoseSoft Incorporated.

SuperKey® is a registered trademark of Borland International, Inc.

Document No. 410610014-500-R00-0787
Part No. 016-014-039

TABLE OF CONTENTS

Introduction

New Features xix
System Requirements xx
About This Manual and Other Assembler Documentation xx
IBM® Compilers and Assemblers xxiii
Books on Assembly Language xxiii
Notational Conventions xxiv
Getting Assistance or Reporting Problems xxviii

Part 1 ◇ Using Assembler Programs

1 Getting Started 5

1.1 Setting Up Your System 7
- 1.1.1 Making Backup Copies 7
- 1.1.2 Choosing a Configuration Strategy 7
- 1.1.3 Copying Files 9
- 1.1.4 Setting Environment Variables 10

1.2 Choosing a Program Type 10

1.3 The Program-Development Cycle 11

1.4 Developing Programs 14
- 1.4.1 Writing and Editing Assembly-Language Source Code 14
- 1.4.2 Assembling Source Files 17
- 1.4.3 Converting Cross-Reference Files 17
- 1.4.4 Creating Library Files 18

1.4.5 Linking Object Files 18
1.4.6 Converting to .COM Format 19
1.4.7 Debugging 20

2 Using MASM 21

2.1 Running the Assembler 23
2.1.1 Assembly Using a Command Line 23
2.1.2 Assembly Using Prompts 25

2.2 Using Environment Variables 26
2.2.1 The INCLUDE Environment Variable 26
2.2.2 The MASM Environment Variable 27

2.3 Controlling Message Output 28

2.4 Using MASM Options 29
2.4.1 Specifying the Segment-Order Method 30
2.4.2 Setting the File-Buffer Size 31
2.4.3 Creating a Pass 1 Listing 32
2.4.4 Defining Assembler Symbols 32
2.4.5 Creating Code for a Floating-Point Emulator 33
2.4.6 Getting Command-Line Help 34
2.4.7 Setting a Search Path for Include Files 35
2.4.8 Specifying Listing and Cross-Reference Files 35
2.4.9 Specifying Case Sensitivity 36
2.4.10 Suppressing Tables in the Listing File 37
2.4.11 Checking for Impure Code 37
2.4.12 Controlling Display of Assembly Statistics 38
2.4.13 Setting the Warning Level 39
2.4.14 Listing False Conditionals 40
2.4.15 Displaying Error Lines on the Screen 41
2.4.16 Writing Symbolic Information to the Object File 41

2.5 Reading Assembly Listings 42
2.5.1 Reading Code in a Listing 42
2.5.2 Reading a Macro Table 45
2.5.3 Reading a Structure and Record Table 45
2.5.4 Reading a Segment and Group Table 46
2.5.5 Reading a Symbol Table 47
2.5.6 Reading Assembly Statistics 49
2.5.7 Reading a Pass 1 Listing 49

CONTENTS

3 Using CREF 51

3.1 Using CREF 53

3.1.1 Using a Command Line to Create a Cross-Reference Listing 53

3.1.2 Using Prompts to Create a Cross-Reference Listing 54

3.2 Reading Cross-Reference Listings 55

Part 2 ◇ Using Directives

4 Writing Source Code 63

4.1 Writing Assembly-Language Statements 65

4.1.1 Using Mnemonics and Operands 66

4.1.2 Writing Comments 67

4.2 Assigning Names to Symbols 67

4.3 Constants 69

4.3.1 Integer Constants 70

4.3.1.1 Specifying Integers with Radix Specifiers ... 70

4.3.1.2 Setting the Default Radix 71

4.3.2 Packed Binary Coded Decimal Constants 72

4.3.3 Real-Number Constants 73

4.3.4 String Constants 74

4.4 Defining Default Assembly Behavior 75

4.5 Ending a Source File 78

5 Defining Segment Structure 81

5.1 Simplified Segment Definitions 83

5.1.1 Understanding Memory Models 84

5.1.2 Specifying DOS Segment Order 85

5.1.3 Defining the Memory Model 87

5.1.4 Defining Simplified Segments 88

5.1.5 Using Predefined Equates 90

5.1.6 Simplified Segment Defaults 92

5.1.7 Default Segment Names 93

5.2 Full Segment Definitions....95
5.2.1 Setting the Segment-Order Method....96
5.2.2 Defining Full Segments....97
5.2.2.1 Controlling Alignment with Align Type....98
5.2.2.2 Setting Segment Word Size with Use Type....98
5.2.2.3 Defining Segment Combinations with Combine Type....100
5.2.2.4 Controlling Segment Structure with Class Type....104
5.3 Defining Segment Groups....106
5.4 Associating Segments with Registers....109
5.5 Initializing Segment Registers....111
5.5.1 Initializing the CS and IP Registers....111
5.5.2 Initializing the DS Register....112
5.5.3 Initializing the SS and SP Registers....114
5.5.4 Initializing the ES Register....115
5.6 Nesting Segments....115

6 Defining Labels and Variables....117

6.1 Using Type Specifiers....119
6.2 Defining Code Labels....120
6.2.1 Near Code Labels....120
6.2.2 Procedure Labels....121
6.2.3 Code Labels Defined with the LABEL Directive....122
6.3 Defining and Initializing Data....123
6.3.1 Variables....123
6.3.1.1 Integer Variables....124
6.3.1.2 Binary Coded Decimal Variables....127
6.3.1.3 String Variables....127
6.3.1.4 Pointer Variables....128
6.3.1.5 Real-Number Variables....130
6.3.2 Arrays and Buffers....135
6.3.3 Labeling Variables....136
6.4 Setting the Location Counter....137
6.5 Aligning Data....138

CONTENTS

7 Using Structures and Records.....141

7.1 Structures 143
7.1.1 Declaring Structure Types 143
7.1.2 Defining Structure Variables 145
7.1.3 Using Structure Operands.....146

7.2 Records 147
7.2.1 Declaring Record Types.....148
7.2.2 Defining Record Variables.....150
7.2.3 Using Record Operands and Record Variables 151
7.2.4 Record Operators.....153
7.2.4.1 The MASK Operator 153
7.2.4.2 The WIDTH Operator 153
7.2.5 Using Record-Field Operands.....154

8 Creating Programs from Multiple Modules.....157

8.1 Declaring Symbols Public.....160

8.2 Declaring Symbols External 161

8.3 Using Multiple Modules.....164

8.4 Declaring Symbols Communal.....165

8.5 Specifying Library Files 169

9 Using Operands and Expressions..171

9.1 Using Operands with Directives 173

9.2 Using Operators 174
9.2.1 Calculation Operators 174
9.2.1.1 Arithmetic Operators 175
9.2.1.2 Structure-Field-Name Operator 176
9.2.1.3 Index Operator.....177
9.2.1.4 Shift Operators 178
9.2.1.5 Bitwise Logical Operators.....179
9.2.2 Relational Operators 180
9.2.3 Segment-Override Operator 181

9.2.4 Type Operators 182
9.2.4.1 PTR Operator 182
9.2.4.2 SHORT Operator 183
9.2.4.3 THIS Operator 183
9.2.4.4 HIGH and LOW Operators 184
9.2.4.5 SEG Operator 184
9.2.4.6 OFFSET Operator 185
9.2.4.7 .TYPE Operator 186
9.2.4.8 TYPE Operator 187
9.2.4.9 LENGTH Operator 188
9.2.4.10 SIZE Operator 188
9.2.5 Operator Precedence 189
9.3 Using the Location Counter 190
9.4 Using Forward References 191
9.4.1 Forward References to Labels 192
9.4.2 Forward References to Variables 194
9.5 Strong Typing for Memory Operands 194
10 Assembling Conditionally 197
10.1 Using Conditional-Assembly Directives 199
10.1.1 Testing Expressions with IF and IFE Directives 200
10.1.2 Testing the Pass with IF1 and IF2 Directives 201
10.1.3 Testing Symbol Definition with IFDEF and IFNDEF Directives 201
10.1.4 Verifying Macro Parameters with IFB and IFNB Directives 202
10.1.5 Comparing Macro Arguments with IFIDN and IFDIF Directives 203
10.2 Using Conditional-Error Directives 204
10.2.1 Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 205
10.2.2 Testing Expressions with .ERRE or .ERRNZ Directives 206
10.2.3 Verifying Symbol Definition with .ERRDEF and .ERRNDEF Directives 207

10.2.4 Testing for Macro Parameters with .ERRB and .ERRNB Directives 207
10.2.5 Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives 208

11 Using Equates, Macros, and Repeat Blocks 211

11.1 Using Equates 213
11.1.1 Redefinable Numeric Equates 213
11.1.2 Nonredefinable Numeric Equates 214
11.1.3 String Equates 216

11.2 Using Macros 217
11.2.1 Defining Macros 218
11.2.2 Calling Macros 219
11.2.3 Using Local Symbols 220
11.2.4 Exiting from a Macro 222

11.3 Defining Repeat Blocks 223
11.3.1 The REPT Directive 223
11.3.2 The IRP Directive 224
11.3.3 The IRPC Directive 225

11.4 Using Macro Operators 226
11.4.1 Substitute Operator 226
11.4.2 Literal-Text Operator 228
11.4.3 Literal-Character Operator 229
11.4.4 Expression Operator 230
11.4.5 Macro Comments 231

11.5 Using Recursive, Nested, and Redefined Macros 231
11.5.1 Using Recursion 231
11.5.2 Nesting Macro Definitions 232
11.5.3 Nesting Macro Calls 233
11.5.4 Redefining Macros 234
11.5.5 Avoiding Inadvertent Substitutions 234

11.6 Managing Macros and Equates 235
11.6.1 Using Include Files 235
11.6.2 Purging Macros from Memory 237

12 Controlling Assembly Output 239
12.1 Sending Messages to the Standard Output Device 241
12.2 Controlling Page Format in Listings 242
12.2.1 Setting the Listing Title 242
12.2.2 Setting the Listing Subtitle 243
12.2.3 Controlling Page Breaks 243
12.3 Controlling the Contents of Listings 245
12.3.1 Suppressing and Restoring Listing Output 245
12.3.2 Controlling Listing of Conditional Blocks 246
12.3.3 Controlling Listing of Macros 247
12.4 Controlling Cross-Reference Output 249

Part 3 ◇ Using Instructions

13 Understanding 8086-Family Processors 255
13.1 Using the 8086-Family Processors 257
13.1.1 Processor Differences 257
13.1.2 Real and Protected Modes 259
13.2 Segmented Addresses 260
13.3 Using 8086-Family Registers 261
13.3.1 Segment Registers 263
13.3.2 General-Purpose Registers 264
13.3.3 Other Registers 266
13.3.4 The Flags Register 266
13.3.5 8087-Family Registers 268
13.4 Using the 80386 Processor Under DOS 269

14 Using Addressing Modes 271
14.1 Using Immediate Operands 273
14.2 Using Register Operands 274

14.3 Using Memory Operands 276
14.3.1 Direct Memory Operands 276
14.3.2 Indirect Memory Operands 278
14.3.3 80386 Indirect Memory Operands 282

15 Loading, Storing, and Moving Data 287
15.1 Transferring Data 289
15.1.1 Copying Data 289
15.1.2 Exchanging Data 290
15.1.3 Looking Up Data 290
15.1.4 Transferring Flags 291
15.2 Converting between Data Sizes 292
15.2.1 Extending Signed Values 292
15.2.2 Extending Unsigned Values 294
15.2.3 Moving and Extending Values 294
15.3 Loading Pointers 295
15.3.1 Loading Near Pointers 295
15.3.2 Loading Far Pointers 296
15.4 Transferring Data to and from the Stack 298
15.4.1 Pushing and Popping 298
15.4.2 Using the Stack 301
15.4.3 Saving Flags on the Stack 301
15.4.4 Saving All Registers on the Stack 302
15.5 Transferring Data to and from Ports 303

16 Doing Arithmetic and Bit Manipulations 305
16.1 Adding 307
16.1.1 Adding Values Directly 307
16.1.2 Adding Values in Multiple Registers 309
16.2 Subtracting 309
16.2.1 Subtracting Values Directly 310
16.2.2 Subtracting with Values in Multiple Registers 311

16.3 Multiplying 312
16.4 Dividing 314
16.5 Calculating with Binary Coded Decimals 316
16.5.1 Unpacked BCD Numbers 317
16.5.2 Packed BCD Numbers 319
16.6 Doing Logical Bit Manipulations 320
16.6.1 AND Operations 321
16.6.2 OR Operations 322
16.6.3 XOR Operations 322
16.6.4 NOT Operations 323
16.7 Scanning for Set Bits 324
16.8 Shifting and Rotating Bits 325
16.8.1 Multiplying and Dividing by Constants 327
16.8.2 Moving Bits to the Least-Significant Position 329
16.8.3 Adjusting Masks 329
16.8.4 Shifting Multiword Values 329
16.8.5 Shifting Multiple Bits 330

17 Controlling Program Flow 331
17.1 Jumping 333
17.1.1 Jumping Unconditionally 333
17.1.2 Jumping Conditionally 335
17.1.2.1 Comparing and Jumping 335
17.1.2.2 Jumping Based on Flag Status 338
17.1.2.3 Testing Bits and Jumping 340
17.1.2.4 Testing and Setting Bits 341
17.2 Looping 343
17.3 Setting Bytes Conditionally 345
17.4 Using Procedures 346
17.4.1 Calling Procedures 347
17.4.2 Defining Procedures 347
17.4.3 Passing Arguments on the Stack 349
17.4.4 Using Local Variables 351
17.4.5 Setting Up Stack Frames 354

17.5 Using Interrupts 355
17.5.1 Calling Interrupts 356
17.5.2 Defining and Redefining Interrupt Routines ... 358
17.6 Checking Memory Ranges 361

18 Processing Strings 363
18.1 Setting Up String Operations 365
18.2 Moving Strings 368
18.3 Searching Strings 370
18.4 Comparing Strings 371
18.5 Filling Strings 373
18.6 Loading Values from Strings 374
18.7 Transferring Strings to and from Ports 375

19 Calculating with a Math Coprocessor 377
19.1 Coprocessor Architecture 379
19.1.1 Coprocessor Data Registers 380
19.1.2 Coprocessor Control Registers 381
19.2 Emulation 382
19.3 Using Coprocessor Instructions 382
19.3.1 Using Implied Operands in the Classical-Stack Form 383
19.3.2 Using Memory Operands 384
19.3.3 Specifying Operands in the Register Form 385
19.3.4 Specifying Operands in the Register-Pop Form 386
19.4 Coordinating Memory Access 387

19.5 Transferring Data 388
19.5.1 Transferring Data to and from Registers 388
19.5.2 Loading Constants 391
19.5.3 Transferring Control Data 392
19.6 Doing Arithmetic Calculations 393
19.7 Controlling Program Flow 399
19.7.1 Comparing Operands to Control Program Flow 401
19.7.2 Testing Control Flags after Other Instructions 404
19.8 Using Transcendental Instructions 404
19.9 Controlling the Coprocessor 406

20 Controlling the Processor 409
20.1 Controlling Timing and Alignment 411
20.2 Controlling the Processor 411
20.3 Controlling Protected-Mode Processes 412
20.4 Controlling the 80386 413

Appendixes

A New Features 417
A.1 MASM Enhancements 419
A.1.1 80386 Support 419
A.1.2 Segment Simplification 420
A.1.3 Performance Improvements 420
A.1.4 Enhanced Error Handling 421
A.1.5 New Options 421
A.1.6 Environment Variables 422
A.1.7 String Equates 422
A.1.8 RETF and RETN Instructions 422
A.1.9 Communal Variables 422
A.1.10 Including Library Files 422
A.1.11 Flexible Structure Definitions 423

A.2 Link Enhancements....423
A.3 The CodeView Debugger....423
A.4 SETENV....424
A.5 Compatibility with Assemblers and Compilers....424

B Error Messages and Exit Codes....427
B.1 MASM Messages and Exit Codes....429
B.1.1 Assembler Status Messages....429
B.1.2 Numbered Assembler Messages....430
B.1.3 Unnumbered Error Messages....446
B.1.4 MASM Exit Codes....448
B.2 CREF Error Messages and Exit Codes....449

Index....451

Figures

Figure 1.1 The Program-Development Cycle 12
Figure 5.1 Segment Structure with Combine and Align Types.....103
Figure 5.2 Segment Structure with Groups.....108
Figure 6.1 Encoding for Real Numbers in IEEE Format.....132
Figure 6.2 Encoding for Real Numbers in Microsoft Binary Format.....133
Figure 6.3 Encoding for Real Numbers in Temporary-Real Format 134
Figure 13.1 Register for 8088–80286 Processors.....262
Figure 13.2 Extended Registers of 80386 Processor.....263
Figure 13.3 Flags for 8088–80386 Processors 267
Figure 15.1 Stack Status after Pushes and Pops.....299
Figure 16.1 Shifts and Rotates.....326
Figure 17.1 Procedure Arguments on the Stack.....351
Figure 17.2 Local Variables on the Stack.....353
Figure 17.3 Operation of Interrupts.....357
Figure 19.1 Coprocessor Data Registers.....380
Figure 19.2 Coprocessor Control Registers.....381
Figure 19.3 Coprocessor and Processor Control Flags.....400

Tables

Table 2.1 Warning Levels 39
Table 2.2 Symbols and Abbreviations in Listings 43
Table 4.1 Reserved Names 69
Table 4.2 Digits Used with Each Radix 70
Table 5.1 Default Segments and Types for Standard Memory Models 93
Table 9.1 Arithmetic Operators 175
Table 9.2 Logical Operators 179
Table 9.3 Relational Operators 180
Table 9.4 .TYPE Operator and Variable Attributes 186
Table 9.5 Operator Precedence 190
Table 10.1 Conditional-Error Directives 204
Table 14.1 Register Operands 275
Table 14.2 Indirect Addressing Modes 279
Table 16.1 Values Returned by Logical Operations 320
Table 17.1 Conditional-Jump Instructions Used after Compare 337
Table 18.1 Requirements for String Instructions 367
Table 19.1 Coprocessor Operand Forms 383
Table 19.2 Control-Flag Settings after Compare or Test 401
Table A.1 80386 and 80387 Instructions 420

INTRODUCTION

Welcome to the Microsoft® Macro Assembler (**MASM**). This package provides all the tools you need to create assembly-language programs.

The Macro Assembler provides a logical programming syntax suited to the segmented architecture of the 8086, 8088, 80186, 80188, 80286, and 80386 microprocessors (8086-family), and the 8087, 80287, and 80387 math coprocessors (8087-family).

The assembler produces relocatable object modules from assembly-language source files. These object modules can be linked using **LINK**, the Microsoft Overlay Linker, to create executable programs for the MS-DOS® operating system. Object modules created with **MASM** are compatible with many high-level-language object modules, including those created with the Microsoft BASIC, C, FORTRAN, and Pascal compilers.

MASM has a variety of standard features that make program development easier:

- It has a full set of macro directives.
- It allows conditional assembly of portions of a source file.
- It supports a wide range of operators for creating complex assembly-time expressions.
- It carries out strict syntax checking of all instruction statements, including strong typing for memory operands.

New Features

This version of the assembler has the following major new features:

- All instructions and addressing modes of the 80386 processor and 80387 coprocessor are now supported.
- The new CodeView® window-oriented debugger allows source-level debugging on assembly-language files and has many other powerful features.
- New segment directives allow simplified segment definitions. These optional directives implement the segment conventions used in Microsoft high-level languages.

- Error messages have been clarified and enhanced.
- The default format for initializing real-number variables has been changed from Microsoft Binary to the more common IEEE (Institute of Electrical and Electronic Engineers, Inc.) format.

Note

In addition to these new features, there are numerous minor enhancements. If you are updating from a previous version of the Microsoft Macro Assembler, you may want to start by reading Appendix A, "New Features." This appendix summarizes new features added for Version 5.0 and discusses compatibility issues.

System Requirements

In addition to a computer with one of the 8086-family processors, you must have Version 2.0 or later of the MS-DOS or IBM® PC-DOS operating system. (Since these two operating systems are essentially the same, this manual uses the term DOS to include both.) To run the assembler itself, your computer system must have approximately 192K (kilobytes) of memory. The CodeView debugger requires approximately 320K. Actual memory requirements vary depending on the DOS version used, the memory used by any resident programs, and the size of the files being assembled or debugged.

About This Manual and Other Assembler Documentation

This manual is intended as a reference manual for writing applications programs in assembly language. It is not intended as a tutorial for beginners, nor does it discuss systems programming or advanced techniques.

This manual is divided into three major parts. Part 1 is called "Using Assembler Programs," and it comprises chapters 1–3. Chapters 4–12 make up Part 2, "Using Directives." The third part, called "Using Instructions," comprises chapters 13–20. Two appendixes follow Part 3.

Important topics for the programmer and their references are listed below:

Information	Location
How to set up the assembler software	Chapter 1, "Getting Started," tells how to set up the assembler and utility software.
An overview of the program-development process	Chapter 1, "Getting Started," describes the program-development process and gives brief examples of each step.
How to use the assembler and the other programs provided with the Microsoft Macro Assembler package	Part 1, "Using Assembler Programs," describes the command lines, options, and output of **MASM** and **CREF**. The Microsoft CodeView and Utilities manual describes the command lines, options, commands, and output of the CodeView debugger, **LINK**, **LIB**, **MAKE**, and other utilities. Error messages are described in Appendix B of the respective manuals. The command-line syntax for all assembler programs is summarized in the *Microsoft Macro Assembler Reference.*
An overview of the format for assembly-language source code	Chapter 1, "Getting Started," shows examples of assembly-language source files, and Chapter 4, "Writing Source Code," (in Part 2) discusses basic concepts in a reference format.
How to program in the version of assembly language recognized by **MASM**	Part 2, "Using Directives," explains the directives, operands, operators, expressions, and other language features understood by **MASM**. However, the manual is not designed to teach novice users how to program in assembly language. If you are new to assembly language, you will still need additional books or courses. Some tutorial books that may be helpful are listed later in this introduction.
An overview of the architecture of 8086-family processors	Chapter 13, "Understanding 8086-Family Processors," (in Part 3) discusses segments, memory use, registers, and other basic features of 8086-family processors.

How to use the instruction sets for the 8086, 80186, 80286, or 80386 microprocessor	Part 3, "Using Instructions," describes each of the instructions. The material is intended as a reference, not a tutorial. Beginners may need to study other books on assembly language.
Reference data on instructions	Another manual in the Macro Assembler package, the *Microsoft Macro Assembler Reference*, lists each instruction alphabetically and gives data on encoding and timing for each. This data is particularly useful for programmers who wish to optimize assembly code.
How to use the instruction sets of the 8087, 80287, or 80387 math coprocessor	Chapter 19, "Calculating with a Math Coprocessor," describes the coprocessor instructions and tells how you can use the most important ones.
Information on DOS structure and function calls	Although this information may be useful to many programmers, it is beyond the scope of the documentation provided with the Microsoft Macro Assembler package. You can find information on DOS in the *Microsoft MS-DOS Programmer's Reference* and in many other books about DOS. Some of the books listed later in this introduction cover these topics.
How to write assembly-language routines for high-level languages	The *Microsoft Mixed-Language Programming Guide* describes the calling and naming conventions of Microsoft high-level languages and tells how to write assembly modules that can be linked with modules created with high-level languages.
Hardware features of your computer	For some assembly-language tasks, you may need to know about the basic input and output systems (BIOS) or other hardware features of the computers that run your programs. Consult the technical manuals for your computer or one of the many books that describe hardware features. Some of the books listed later in this introduction discuss hardware features of IBM and IBM-compatible computers.

IBM® Compilers and Assemblers

Many IBM languages are produced for IBM by Microsoft. IBM languages similar to corresponding Microsoft languages include the following:

IBM Personal Computer Macro Assembler, Versions 1.0 and 2.0

IBM Personal Computer FORTRAN, Version 3.*x*

IBM Personal Computer C, Version 1.0

IBM Personal Computer Pascal, Versions 1.0 to 3.*x*

IBM Personal Computer BASIC Compiler, Versions 1.0 and 2.0

These languages are compatible with the Microsoft Macro Assembler Version 5.0, except as noted in Appendix A, "New Features."

Books on Assembly Language

The following books may be useful in learning to program in assembly language:

Duncan, Ray. *Advanced MS-DOS.* Redmond, Wash.: Microsoft Corporation, 1986.

An intermediate book on writing C and assembly-language programs that interact with MS-DOS (includes DOS and BIOS function descriptions)

Intel Corporation. *iAPX 386 Programmer's Reference Manual.* Santa Clara, Calif. 1986.

Reference manual for 80386 processor and instruction set (manuals for previous processors are also available)

Jourdain, Robert. *Programmer's Problem Solver for the IBM PC, XT and AT.* New York: Brady Communications Company, Inc., 1986.

Reference of routines and techniques for interacting with hardware devices through DOS, BIOS, and ports (high-level routines in BASIC and low- or medium-level routines in assembler)

Lafore, Robert. *Assembly Language Primer for the IBM PC & XT.* New York: Plume/Waite, 1984.

An introduction to assembly language, including some information on DOS function calls and IBM-type BIOS

Metcalf, Christopher D., and Sugiyama, Marc B. *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC & PCjr.* Greensboro, N.C.: COMPUTE! Publications, Inc., 1985.

Beginning discussion of assembly language, including information on the instruction set and MS-DOS function calls

Microsoft. *Microsoft MS-DOS Programmer's Reference.* Redmond, Wash. 1986, 1987.

Reference manual for MS-DOS

Morgan, Christopher, and the Waite Group. *Bluebook of Assembly Routines for the IBM PC.* New York: New American Library, 1984.

Sample assembly routines that can be integrated into assembly or high-level-language programs

Norton, Peter. *The Peter Norton Programmer's Guide to the IBM PC.* Redmond, Wash.: Microsoft Press, 1985.

Information on using IBM-type BIOS and MS-DOS function calls

Scanlon, Leo J. *IBM PC Assembly Language: A Guide for Programmers.* Bovie, Md.: Robert J. Brady Co., 1983.

An introduction to assembly language, including information on DOS function calls

Schneider, Al. *Fundamentals of IBM PC Assembly Language.* Blue Ridge Summit, Pa.: Tab Books Inc., 1984.

An introduction to assembly language, including information on DOS function calls

These books are listed for your convenience only. Microsoft Corporation does not endorse these books (with the exception of those published by Microsoft) or recommend them over others on the same subjects.

Notational Conventions

This manual uses the notation described in the following list.

Example of Convention	Description of Convention
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen or by your printer. For example, the following

source line is printed in this special typeface:

```
mov     ax,WORD PTR string[3]
```

When discussing this source line in text, items appearing on the line, such as `string[3]`, also appear in the special typeface.

Program
.
.
.
Fragment

A column of dots in syntax lines and program examples shows that a portion of the program has been omitted.

For example, in the following program fragment, only the opening lines and the closing lines of a macro are shown. The internal lines are omitted since they are not relevant to the concept being illustrated.

```
work  MACRO   realarg,testarg
      .ERRB  <realarg>  ;; Too few
      .ERRNB <testarg>  ;; Too many
      .                 ;; Just right
      .
      .
      ENDM
```

KEY TERMS

Bold letters indicate command line options, assembly-language keywords or symbols, and the names of files that come with the Microsoft Macro Assembler package.

For instance, the directive **ORG**, the instruction **MOV**, the register **AX**, the option **/ZI**, and the file name **MASM** are always shown in bold when they appear in text or in syntax displays (but not in examples).

In syntax displays, bold type indicates any words, punctuation, or symbols (such as commas, parentheses, semicolons, hyphens, equal signs, or operators) that you must type exactly as shown.

For example, the syntax of the **IFDIF** directive is shown as follows:

IFDIF <*argument1*>,<*argument2*>

The word **IFDIF**, the angle brackets, and the comma are all shown in bold. Therefore they must be typed exactly as shown.

placeholders

Words in italics are placeholders for variable information that you must supply. For example, the syntax of the **OFFSET** operator is shown below:

OFFSET *expression*

This indicates that any *expression* may be supplied following the **OFFSET** operator. When writing source code to match this syntax, you might type

```
OFFSET here+6
```

in which `here+6` is the expression. The placeholder is shown in italics both in syntax displays and in descriptions explaining syntax displays.

[[*optional items*]]

Double brackets surround optional syntax elements. For example, the syntax of the index operator is shown as follows:

[[*expression1*]][*expression2*]

This indicates that *expression1* is optional, since it is contained in double brackets, but *expression2* is required and must be enclosed in brackets.

When writing code to match this syntax, you might type `[bx]`, leaving off the optional *expression1*, or you might type `test[5]`, using `test` as *expression1*.

{*choice1* | *choice2*}

Braces and vertical bars indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items.

For example, the **/W** (warning-level) option has the following syntax:

/W{0 | 1 | 2}

You can type `/W0`, `/W1`, or `/W2` to indicate the desired level of warning. However, typing `/W3` is illegal since 3 is not one of the choices enclosed in braces.

Repeating elements...

Three dots following an item indicate that more items having the same form may be entered.

For example, the syntax of the **PUBLIC** directive is shown below:

PUBLIC *name* [[,*name*]]...

The dots following the second name indicate that you can enter as many *names* as you like as long as each is preceded by a comma. However, since the first *name* is not in brackets, you must enter at least one *name*.

Defined terms and "Prompts"

Quotation marks set off terms defined in the text. For example, the term "indeterminate" appears in quotation marks the first time it is defined.

Quotation marks also set off command-line prompts in text. For example, one **LINK** prompt would be described in text as the "object modules" prompt.

KEY NAMES

Small capital letters are used for the names of keys and key sequences that you must press. Examples include ENTER and CONTROL+C.

■ Example

The following example shows how this manual's notational conventions are used to indicate the syntax of the **MASM** command line:

MASM [[*options*]] *sourcefile* [[,[[*objectfile*]] [[,[[*listingfile*]] [[,[[*crossreferencefile*]]]]]] [[;]]

This syntax shows that you must first type the program name, **MASM**. You can then enter any number of *options*. You must enter a *sourcefile*. You can enter an *objectfile* preceded by a comma. You can enter a *listingfile*, but if you do, you must precede it with the commas associated with the *sourcefile* and *objectfile*. Similarly, you can enter a *crossreferencefile*, but if you do, you must precede it with the commas associated with the other files. You can also enter a semicolon at any point after the *sourcefile*.

For example, any of the following command lines would be legal:

```
MASM test.asm;
MASM /ZI test.asm;
MASM test.asm,,test.lst;
MASM test.asm,,,test.crf
MASM test.asm,test.obj,test.lst,test.crf
MASM test.asm,,,;
```

Getting Assistance or Reporting Problems

If you need help or you feel you have discovered a problem in the software, please provide the following information to help us locate the problem:

- The assembler version number (from the logo that is printed when you invoke the assembler with **MASM**)
- The version of DOS you are running (use the DOS **VER** command)
- Your system configuration (the type of machine you are using, its total memory, and its total free memory at assembler execution time, as well as any other information you think might be useful)
- The assembly command line used (or the link command line if the problem occurred during linking)
- Any object files or libraries you linked with if the problem occurred at link time.

If your program is very large, please try to reduce its size to the smallest possible program that still produces the problem.

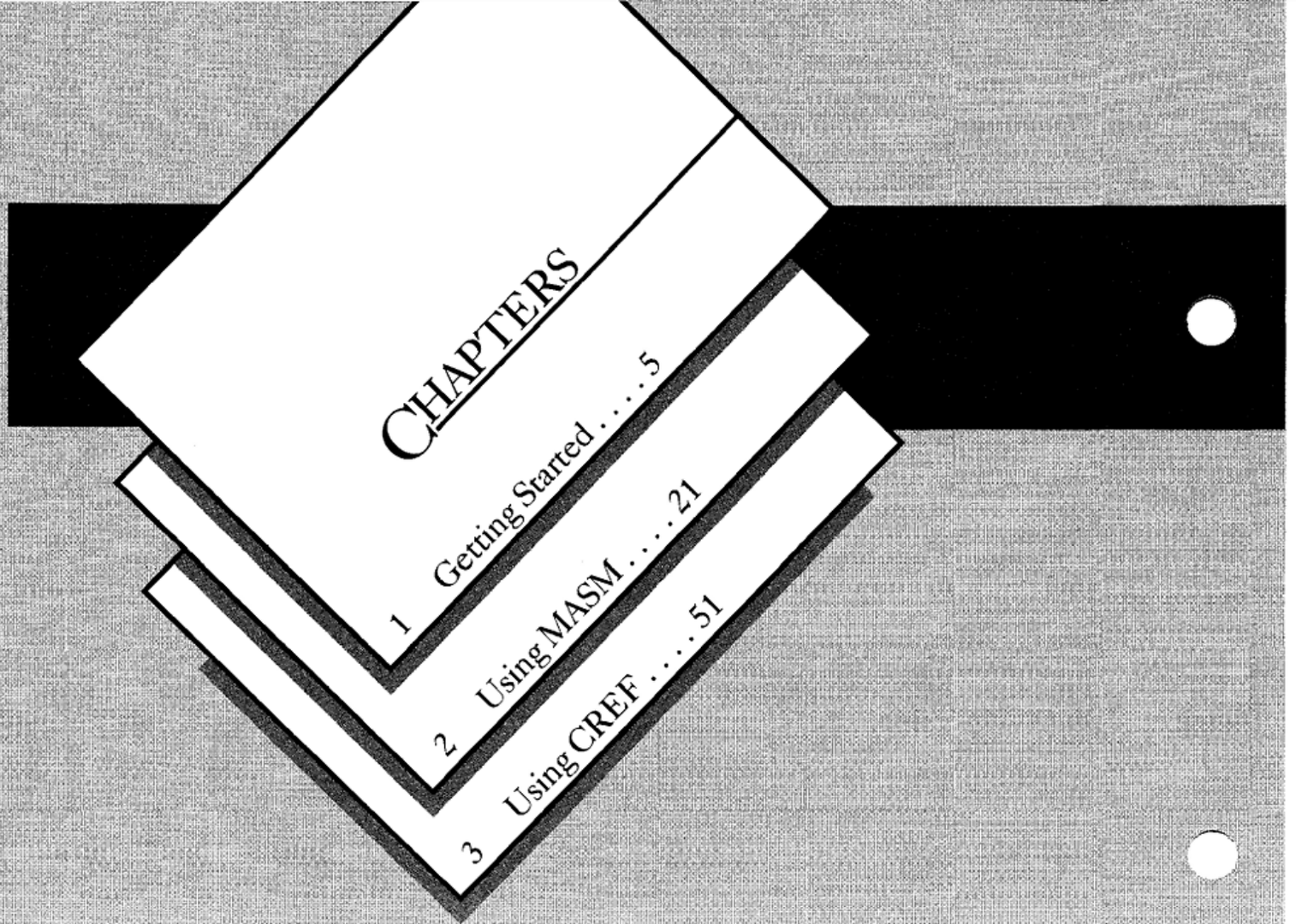
Use the Product Assistance Request form at the back of this manual to send this information to Microsoft.

If you have comments or suggestions regarding any of the manuals accompanying this product, please indicate them on the Document Feedback card at the back of this manual.

You should also fill out and return the Registration Card if you wish to be informed of updates and other information about the assembler.

PART 1

USING ASSEMBLER PROGRAMS


CHAPTERS
1 Getting Started 5
2 Using MASM 21
3 Using CREF 51

PART 1 ◇ USING ASSEMBLER PROGRAMS

Part 1 of the Programmer's Guide (comprising chapters 1–3) summarizes the process of creating programs from assembly-language source files.

Chapter 1 tells you how to set up an efficient system for producing programs. It also gives you examples of simple assembly-language source files and a quick summary of each of the utility programs used in program development.

Chapters 2 and 3 describe the two assembler programs, **MASM** and **CREF**, in detail. Other utility programs, such as **LINK**, **LIB**, **MAKE**, and the CodeView debugger, are described in the Microsoft CodeView and Utilities manual.

CHAPTER 1

GETTING STARTED

1.1 Setting Up Your System 7
1.1.1 Making Backup Copies 7
1.1.2 Choosing a Configuration Strategy 7
1.1.3 Copying Files 9
1.1.4 Setting Environment Variables 10
1.2 Choosing a Program Type 10
1.3 The Program-Development Cycle 11
1.4 Developing Programs 14
1.4.1 Writing and Editing Assembly-Language Source Code 14
1.4.2 Assembling Source Files 17
1.4.3 Converting Cross-Reference Files 17
1.4.4 Creating Library Files 18
1.4.5 Linking Object Files 18
1.4.6 Converting to .COM Format 19
1.4.7 Debugging 20

This chapter tells you how to set up Microsoft Macro Assembler files and to start writing assembly-language programs. It gives an overview of the development process and shows examples using simple programs. It also refers you to the chapters where you can learn more about each subject.

1.1 Setting Up Your System

After opening the Microsoft Macro Assembler package, you should take these four setup steps before you begin developing assembler programs:

1. Make backup copies of the disks in the assembler package.
2. Choose a configuration strategy.
3. Copy the assembler files to the appropriate disks and directories.
4. Set environment variables.

1.1.1 Making Backup Copies

You should make backup copies of the assembler disks before attempting to use any of the programs in the package. Put the copies in a safe place and use them only to restore the originals if they are damaged or destroyed.

All the files on the disks are listed in the file **PACKING.LST** on Disk 1.

The files on the disk are not copy protected. You may make one backup copy for your own use. You may not distribute any executable, object, or library files on the disk. The sample programs are in the public domain.

No license is required to distribute executable files that are created with the assembler.

1.1.2 Choosing a Configuration Strategy

There are several kinds of files on the distribution disk. You can arrange these files in a variety of ways. The two most important considerations are whether or not you have a hard disk and whether you want to use environment variables.

Program development can be affected by the environment variables described below:

Variable	Description
PATH	Specifies the directories where DOS looks for executable files. A common setup with language products is to place executable files in the directory **\BIN** and include this directory in the **PATH** environment string.
LIB	Specifies the directory where **LINK** looks for library and object files. A common setup with language products is to put library and object files in the directory **\LIB** and include this directory in the **LIB** environment string.
INCLUDE	Specifies the directory where **MASM** looks for include files. A common setup with language products is to put macro files and other include files in the directory **\INCLUDE** and to put this directory in the **INCLUDE** environment string.
MASM	Specifies default options that **MASM** uses on start-up.
LINK	Specifies default options that **LINK** uses on start-up.
TMP	Specifies the directory where **LINK** places temporary files if it needs to create them.
INIT	Specifies the directory where **MAKE** looks for the file **TOOLS.INI**, which may contain inference rules. See the documentation of **MAKE** in the Microsoft CodeView and Utilities manual for information on inference rules.

If you have a hard disk, you will probably want to use environment variables to specify locations for library, macro, and executable files. If you do not have a hard disk, you may prefer to leave all files in the root directory.

If you already have other language products on a hard disk, you should consider how your assembler setup interacts with your other languages.

Some users may prefer to have separate directories for library and include files for each language. Others may prefer to have all library and include files in the same directories. If you want all language files in the same directories, make sure you do not have any files with the same names as the ones provided with the Microsoft Macro Assembler.

If you have 5 1/4-inch disks, you will not be able to get all the tools you need for assembly-language development on one disk. A typical setup is shown below:

Disk	Files
1	Source, object, library, and macro files on Disk 1 with (a) source and working object files in the root directory, (b) library and standard object files in directory **\LIB**, and (c) macro files in directory **\INCLUDE**.
2	Executable files for developing programs on Disk 2. This could include **MASM**, **LINK**, a text editor, and possibly **MAKE**, **LIB**, or **CREF**. These files may not all fit on a standard 360K disk, so you will have to decide which are most important for you.
3	The CodeView debugger and any additional utilities on Disk 3.

With this setup, you could keep Disk 1 in Drive A. Then swap Disks 2 and 3 depending on whether you are developing programs or debugging.

1.1.3 Copying Files

A setup batch file called **SETUP.BAT** is provided on Disk 1. You can run it to copy automatically the assembler files to your work disk. The setup program will ask for information about your system and how you want to set it up. Before copying anything to your system, the setup program tells you what it is about to do and prompts for your confirmation.

If you prefer, you can ignore the setup program and copy the files yourself. See the **PACKING.LST** file for a list of files.

Warning

If you have previous versions of the assembler or other programs such as **LINK**, **LIB**, or **MAKE**, you may want to make backup copies or rename the old files so that you do not overwrite them with the new versions.

1.1.4 Setting Environment Variables

If you wish to use environment variables to establish default file locations and options, you will probably want to set the environment variables in your **AUTOEXEC.BAT** or other batch files. The setup program does not attempt to set any environment variables, so you must modify any batch files yourself.

The following lines could be added for a typical hard-disk setup:

```
PATH C:\BIN
SET LIB=C:\LIB
SET INCLUDE=C:\INCLUDE
SET MASM=/ZI
SET LINK=/CO
```

The following lines might be used for the floppy-disk setup described in Section 1.1.2:

```
PATH B:\;A:\
SET LIB=A:\LIB
SET INCLUDE=A:\INCLUDE
SET MASM=/ZI
SET LINK=/CO
```

1.2 Choosing a Program Type

MASM can be used to create different kinds of program files. The source-code format is different for each kind of program. The primary formats are described below:

Type	Description
.EXE	The **.EXE** format is the most common format for programs that will execute under DOS. In future versions of DOS, a similar **.EXE** format will be the only format available for stand-alone programs that take advantage of multitasking. Programs in the **.EXE** format can have multiple segments and can be of any size. Modules can be created and linked using either the assembler or most high-level-language compilers, including all the Microsoft compilers. Modules created in different languages can be

combined into a single program. This is the format recommended by Microsoft for programs of significant size and purpose. The source format for creating this kind of program is described and illustrated throughout the rest of the manual.

.COM The **.COM** format is sometimes convenient for small programs.

Programs in this format are limited to one segment. They can be no larger than 64K (unless they use overlays). They have no file header and are thus smaller than comparable **.EXE** files. This makes programs in the **.COM** format a good choice for small stand-alone assembler programs of several thousand bytes or less. One disadvantage of the **.COM** format is that executable files cannot contain symbolic and source line information for the CodeView debugger. You can only debug **COM** in assembly mode. The source format for **.COM** programs is illustrated briefly in this chapter and described fully in the *Microsoft MS-DOS Programmer's Reference Guide.*

Binary files — Binary files are used for procedures that will be called by the Microsoft and IBM BASIC interpreters.

They are also used by some non-Microsoft compilers. See the manual for the language you are using for details on preparing source files.

Device drivers — Device drivers that set up and control I/O for hardware devices can be developed with the assembler.

The source format for device drives is described in the *Microsoft MS-DOS Programmer's Reference.*

Code for ROMs — The assembler can be used to prepare code that is downloaded to programmable ROM chips. The format is usually a binary format. Methods of translating binary files into a format that can be used in ROM chips vary.

1.3 The Program-Development Cycle

The program-development cycle for assembly language is illustrated in Figure 1.1.

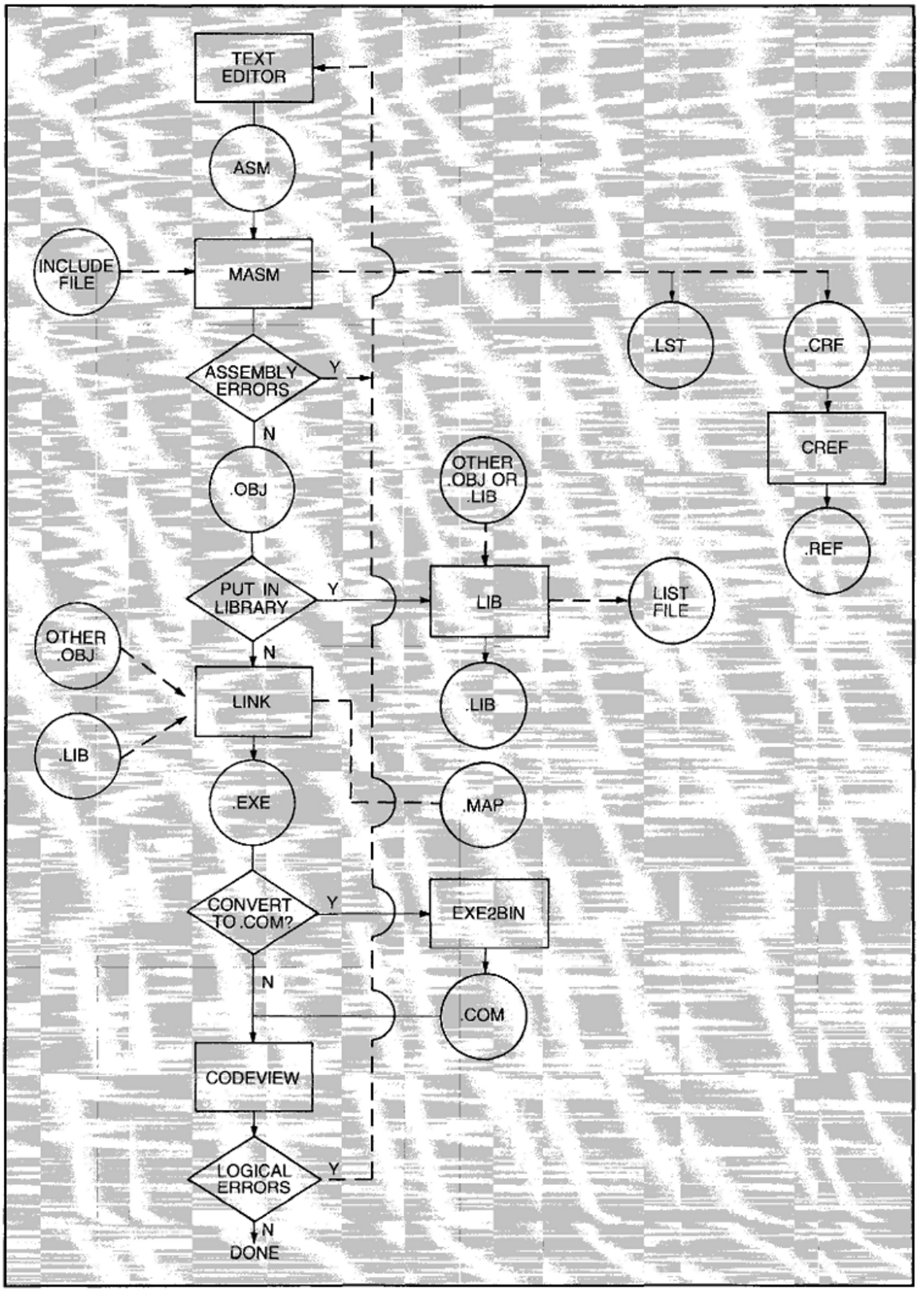


Figure 1.1 The Program-Development Cycle

The specific steps for developing a stand-alone assembler program are listed below:

1. Use a text editor to create or modify assembly-language source modules. By convention, source modules are given the extension **.ASM**. Source modules can be organized in a variety of ways. For instance, you can put all the procedures for a program into one large module, or you can split the procedures between modules. If your program will be linked with high-level-language modules, the source code for these modules is also prepared at this point.

2. Use **MASM** to assemble each of the modules for the program. **MASM** may optionally read in code from include files during assembly. If assembly errors are encountered in a module, you must go back to Step 1 and correct the errors before continuing. For each source (**.ASM**) file, **MASM** creates an object file with the default extension **.OBJ**. Optional listing (**.LST**) and cross-reference (**.CRF**) files can also be created during assembly. If your program will be linked with high-level-language modules, the source modules are compiled to object files at this point.

3. Optionally use **LIB** to gather multiple object files (**.OBJ**) into a single library file having the default extension **.LIB**. It is generally used for object files that will be linked with several different programs. An optional library list file can also be created with **LIB**.

4. Use **LINK** to combine all the object files and library modules that make up a program into a single executable file (with the default extension **.EXE**). An optional **.MAP** file can also be created.

5. Use **EXE2BIN** to convert executable files to a binary format if necessary. It is necessary for programs in the **.COM** format and for binary files that will be read into an interpreter or compiler. Skip this step for programs in the **.EXE** format.

6. Debug your program to discover logical errors. Debugging may involve several techniques, including the following:

   - Running the program and studying its input and output
   - Studying source and listing files
   - Using **CREF** to create a cross-reference-listing (**.REF**) file
   - Using CodeView (**CV**) to debug during execution

   If logical errors are discovered, you must return to Step 1 to correct the source code.

All or part of the program-development cycle can be automated by using **MAKE** with make description files. **MAKE** is most useful for developing complex programs involving numerous source modules. Ordinary DOS batch files may be more efficient for developing single-module programs.

1.4 Developing Programs

The text below takes you through the steps involved in developing programs. Examples are shown for each step. The chapters and manuals that describe each topic in detail are cross-referenced.

1.4.1 Writing and Editing Assembly-Language Source Code

Assembly-language programs are created from one or more source files. Source files are text files that contain statements defining the program's data and instructions.

To create assembly-language source files, you need a text editor capable of producing ASCII (American Standard Code for Information Interchange) files. Lines must be separated by a carriage-return–line-feed combination. If your text editor has a programming or nondocument mode for producing ASCII files, use that mode.

The following examples illustrate source code that produces stand-alone executable programs. Example 1 creates a program in the **.EXE** format, and Example 2 creates the same program in the **.COM** format.

If you are a beginner to assembly language, you can start experimenting by copying these programs. Use the segment shell of the programs, but insert your own data and code.

■ Example 1

```
                TITLE   hello
                DOSSEG
                .MODEL  SMALL①          ; Use Microsoft segment conventions
                                        ;   conventions and small model

                .STACK  100h②           ; Allocate 256-byte stack

                .DATA③
message         DB      "Hello, world.",13,10  ; Message to be written
lmessage        EQU     $ - message            ; Length of message

            ④.CODE
start:      ④mov    ax,@DATA⑤          ; Load segment location
                mov     ds,ax           ;   into DS register

                mov     bx,1            ; Load 1 - file handle for
                                        ;   standard output
              ⑥mov      cx,lmessage     ; Load length of message
              ⑥mov      dx,OFFSET message ; Load address of message
                mov     ah,40h          ; Load number for DOS Write function
                int     21h             ; Call DOS

                mov     ax,4C00h        ; Load DOS Exit function (4Ch)
              ⑦                         ;   in AH and 0 errorlevel in AL
                int     21h             ; Call DOS

                END     start④
```

Note the following points about the source file:

1. The **.MODEL** and **DOSSEG** directives tell **MASM** that you intend to use the Microsoft order and name conventions for segments. These statements automatically define the segments in the correct order and specify **ASSUME** and **GROUP** statements. You can then place segments in your source file in whatever order you find convenient using the **.STACK**, **.DATA**, **.CODE**, and other segment directives. These simplified segment directives are a new feature of Version 5.0. They are optional; you can still define the segments completely by using the directives required by earlier versions of **MASM**. The simplified segment directives and the Microsoft naming conventions are explained in Section 5.1.

2. A stack of 256 (100 hexadecimal) bytes is defined by using the **.STACK** directive. This is an adequate size for most small programs. Programs with many nested procedures may require a larger stack. See Sections 5.1.4, "Defining Simplified Segments," and 5.2.2, "Defining Full Segments," for more information on defining a stack.

3. The **.DATA** directive marks the start of the data segment. A string variable and its length are defined in this segment.

4. The instruction label `start` in the code segment follows the **.CODE** directive and marks the start of the program instructions. The same label is used after the **END** statement to define the point where program execution will start. See Sections 4.5, "Ending a Source File," and 5.5.1, "Initializing the CS and IP Registers," for more information on using the **END** statement and defining the execution starting point.

5. The first two code instructions load the address of the data segment into the **DS** register. The symbol @DATA is an equate representing the name of the segment created with the **.DATA** directive. Predefined segment equates are explained in Section 5.1.5. The **DS** register must always be initialized for source files in the **.EXE** format. Section 5.5 tells how each segment is initialized.

6. The string variable defined earlier is displayed using DOS function 40h (where "h" stands for hexadecimal). File handle 1 (the predefined handle for standard output) is specified to display to the screen. Strings can also be displayed using function 09h. See the *Microsoft MS-DOS Programmer's Reference* or other DOS reference books for more information on DOS calls.

7. DOS function 4Ch is used to terminate the program. While there are other techniques for returning to DOS, this is the one recommended by Microsoft.

The following example shows source code that can be used to create the same program shown earlier, but in the **.COM** format:

■ Example 2

```
            TITLE   hello

_TEXT       SEGMENT①                    ; Define code segment
            ASSUME  cs:_TEXT,ds:_TEXT,ss:_TEXT②
            ORG     100h③               ; Set location counter to 256

start:      jmp     begin④              ; Jump over data

message     DB      "Hello, world.",13,10    ; Message to be written
lmessage    EQU     $ - message              ; Length of message

begin:      mov     bx,1                ; Load 1 - file handle for
                                        ;   standard output
            mov     cx,lmessage         ; Load length of message
            mov     dx,OFFSET message   ; Load address of message
            mov     ah,40h              ; Load number for DOS Write function
            int     21h                 ; Call DOS

            mov     ax,4C00h            ; Load DOS Exit function (4Ch)
                                        ;   in AH and 0 errorlevel in AL
            int     21h                 ; Call DOS
                                        ; Data could be placed here
_TEXT       ENDS①
            END     start
```

Note the following points in which **.COM** programs differ from **.EXE** programs:

1. The **.MODEL** directive cannot be used to define default segments for **.COM** files. However, segment definition is easy, since only one segment can be used. The align, combine, and class types need not be given, since they make no difference for **.COM** files.
2. All segment registers are initialized to the same segment by using the **ASSUME** directive. This tells the assembler which segment to associate with each segment register. See Section 5.4, "Associating Segments with Registers," for more information on the **ASSUME** directive.
3. The **ORG** directive must be used to start assembly at byte 256 (100h). This leaves room for the DOS Program Segment Prefix, which is automatically loaded into memory at run time. See Section 6.4, "Setting the Location Counter," for information on how the **ORG** directive changes the location counter.
4. Although any program data must be included in the single segment, it must not be executed. You can use the **JMP** instruction to skip over data (as shown in the example) or you can put the data at the end after the program returns to DOS.

1.4.2 Assembling Source Files

Source modules are assembled with **MASM**. The **MASM** command-line syntax is shown below:

MASM [*options*] *sourcefile* [,[*objectfile*] [,[*listingfile*] [,[*crossreferencefile*]]]] [;]

Assume you had an assembly source file called `hello.asm`. For the fastest possible assembly, you could start **MASM** with the following command line:

```
MASM hello;
```

The output would be an object file called `hello.obj`. To assemble the same source file with the maximum amount of debugging information, use the following command line:

```
MASM /V /Z /ZI hello,,,;
```

The **/V** and **/Z** options instruct **MASM** to send additional statistics and error information to the screen during assembly. The **/ZI** option instructs **MASM** to include the information required by the CodeView debugger in the object file. The output of this command is three files: the object file `hello.obj`, the listing file `hello.lst`, and the cross-reference file `hello.crf`.

Chapter 2, "Using MASM," describes the **MASM** command line, options, and listing format in more detail.

1.4.3 Converting Cross-Reference Files

Cross-reference files produced by **MASM** are in a binary format and must be converted using **CREF**. The command-line syntax is shown below:

CREF *crossreferencefile* [,*crossreferencelisting*] [;]

To convert the cross-reference file `hello.crf` into an ASCII file that cross-references symbols that are used in `hello.asm`, use the following command line:

```
CREF hello;
```

The output file is called `hello.ref`.

The **CREF** command line and listing format are described in Chapter 3, "Using CREF."

1.4.4 Creating Library Files

Object files created with **MASM** or with Microsoft high-level-language compilers can be converted to library files by using **LIB**. The command-line syntax is shown below:

LIB *oldlibrary* [[**/PAGESIZE:***number*]] [[*commands*]] [,[[*listfile*]] [,[[*newlibrary*]]]] [[;]]

For example, assume you had used **MASM** to assemble two source files containing graphics procedures and you want to be able to call the procedures from several different programs. The object files containing the procedures are called `dots.obj` and `lines.obj`.

You could combine these files into a file called `graphics.lib` using the following command line:

```
LIB graphics +dots +lines;
```

If you later wanted to add another object file called `circles.obj` and at the same time get a listing of the procedures in the library, you could use the following command line:

```
LIB graphics +circles,graphics.lst
```

The **LIB** command line, commands, and listing format are explained in the Microsoft CodeView and Utilities manual.

1.4.5 Linking Object Files

Object files are linked into executable files using **LINK**. The **LINK** command-line syntax is shown below:

LINK [[*options*]] *objectfiles* [,[[*executablefile*]] [,[[*mapfile*]] [,[[*libraryfiles*]]]] [[;]]

Assume you want to create an executable file from the single module `hello.obj`. The source file was written for the **.EXE** format (see Section 1.4.1, "Writing and Editing Assembly-Language Source Code") and was assembled using the **/ZI** option. You plan to debug the program with the CodeView debugger. Use the following command line:

```
LINK /CO hello;
```

The output file is `hello.exe`. It contains symbolic and line-number information for the debugger. The file can now be run from the DOS command line or from within the CodeView debugger.

After you have debugged the program, you will probably want to create a final version with no symbolic information. To do so, use the following command line:

```
LINK hello;
```

This command line could also be used if the source file had been prepared in the **.COM** Format. However, in that case the output file `hello.exe` could not be run. Another step is required, as described in Section 1.4.6, "Converting to **.COM** Format."

Now assume that you want to create a large program called `picture.exe` that has two object files (`picture` and `picture2`) and calls external procedures from the library file described in Section 1.4.4, "Creating Library Files." Use the following command line:

```
LINK /CO picture picture2,,,graphics;
```

The library file `graphics.lib` would need to be in the current directory or in the directory described by the **LIB** environment variable. The procedure calls would have to be declared external in the source file, as described in Section 8.2, "Declaring Symbols External."

The **LINK** options, command line, and listing format are described in the Microsoft CodeView and Utilities manual.

1.4.6 Converting to .COM Format

Source files prepared in the **.COM** format require an additional conversion step after linking. The program that does the conversion is called **EXE2BIN**. It is not included in the Macro Assembler package, but it does come with the MS-DOS and PC-DOS operating systems. The syntax is shown below:

EXE2BIN *exefile* [[*binaryfile*]]

To convert a file called `hello.exe` to an executable file called `hello.com`, use the following command line:

```
EXE2BIN hello hello.com
```

Note that you must specify the extension **.COM**, since **BIN** is the default extension. The **.EXE** file must have been prepared from source and object files in the valid **.COM** format.

EXE2BIN can also be used to prepare binary files for use with the Microsoft or IBM BASIC interpreters. See the BASIC interpreter manual for more information.

1.4.7 Debugging

The CodeView debugger is usually the most efficient tool for debugging assembler programs. The command-line syntax is shown below:

CV [[*options*]] *executablefile* [[*arguments*]]

To debug a program called `hello.exe` on an IBM Personal Computer, use the following command line:

```
CV hello
```

Note that in order for the debugger to display symbolic information, the program should have been assembled with the **/ZI** option and linked with the **/CO** option. Additional CodeView options may be required for other situations. For instance, graphics programs always require the **/S** option. To debug a graphics program called `circles.com` on an IBM-compatible computer, use the following command line:

```
CV /W/I/S circles.com
```

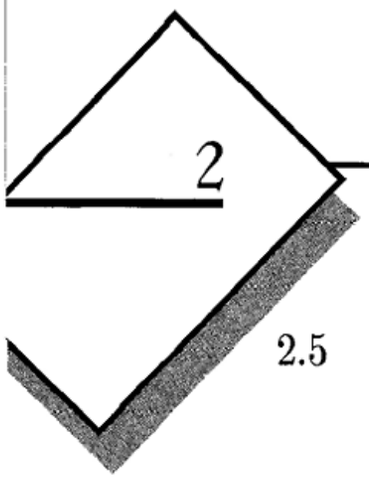
The **/W** and **/I** options tell the debugger to use IBM-compatible features. Note that the **.COM** extension must be specified, since the debugger assumes files without extensions are **.EXE** files.

For information about CodeView command lines, options, and commands, see the Microsoft CodeView and Utilities manual.

CHAPTER 2

USING MASM

2.1 Running the Assembler 23
2.1.1 Assembly Using a Command Line 23
2.1.2 Assembly Using Prompts 25
2.2 Using Environment Variables 26
2.2.1 The INCLUDE Environment Variable 26
2.2.2 The MASM Environment Variable 27
2.3 Controlling Message Output 28
2.4 Using MASM Options 29
2.4.1 Specifying the Segment-Order Method 30
2.4.2 Setting the File-Buffer Size 31
2.4.3 Creating a Pass 1 Listing 32
2.4.4 Defining Assembler Symbols 32
2.4.5 Creating Code for a Floating-Point Emulator 33
2.4.6 Getting Command-Line Help 34
2.4.7 Setting a Search Path for Include Files 35
2.4.8 Specifying Listing and Cross-Reference Files 35
2.4.9 Specifying Case Sensitivity 36
2.4.10 Suppressing Tables in the Listing File 37
2.4.11 Checking for Impure Code 37
2.4.12 Controlling Display of Assembly Statistics 38
2.4.13 Setting the Warning Level 39
2.4.14 Listing False Conditionals 40
2.4.15 Displaying Error Lines on the Screen 41
2.4.16 Writing Symbolic Information to the Object File 41



2.5 Reading Assembly Listings 42
2.5.1 Reading Code in a Listing 42
2.5.2 Reading a Macro Table 45
2.5.3 Reading a Structure and Record Table 45
2.5.4 Reading a Segment and Group Table 46
2.5.5 Reading a Symbol Table 47
2.5.6 Reading Assembly Statistics 49
2.5.7 Reading a Pass 1 Listing 49

The Microsoft Macro Assembler (**MASM**) assembles 8086, 80186, 80286, and 80386 assembly-language source files and creates relocatable object files. Object files can then be linked to form an executable file.

This chapter tells you how to run **MASM**, explains the options and environment variables that control its behavior, and describes the format of the assembly listings it generates.

2.1 Running the Assembler

You can assemble source files with **MASM** by using two different methods: by giving a command line at the DOS prompt or by responding to a series of prompts.

Once you have started **MASM**, it attempts to process the source file you specified. If errors are encountered, they are output to the screen and **MASM** terminates. If no errors are encountered, **MASM** outputs an object file. It can also output listing and cross-reference files if they are specified. You can terminate **MASM** at any time by pressing CONTROL-C or CONTROL-BREAK.

2.1.1 Assembly Using a Command Line

You can assemble a program source file by typing the **MASM** command name and the names of the files you wish to process. The command line has the following form:

MASM [[*options*]] *sourcefile* [[,[[*objectfile*]] [[,[[*listingfile*]] [[,[[*crossreferencefile*]]]]]] [[;]]

The *options* can be any combination of the assembler options described in Section 2.4. The option letter or letters must be preceded by a forward slash (/) or a dash (-). Examples in this manual use a forward slash. The forward slash and dash characters cannot be mixed in the same command line. Although shown at the beginning of the syntax line above, options may actually be placed anywhere on the command line. An option affects all relevant files in the command line even if the option appears at the end of the line.

The *sourcefile* must be the name of the source file to be assembled. If you do not supply a file-name extension, **MASM** supplies the extension **.ASM**.

The optional *objectfile* is the name of the file to receive the relocatable object code. If you do not supply a name, **MASM** uses the source-file name, but replaces the extension with **.OBJ**.

The optional *listingfile* is the name of the file to receive the assembly listing. The assembly listing shows the assembled code for each source statement and for the names and types of symbols defined in the program. If you do not supply a file-name extension, the Macro Assembler supplies the extension **.LST**.

The optional *crossreferencefile* is the name of the file to receive the cross-reference output. The resulting cross-reference file can be processed with **CREF**, the Microsoft Cross-Reference Utility, to create a cross-reference listing of the symbols in the program. The cross-reference listing can be used for program debugging. If you do not supply a file-name extension, **MASM** supplies **.CRF** by default.

You can use a semicolon (;) anywhere after the *sourcefile* to select defaults for the remaining file names. A semicolon after the source-file name selects a default object-file name and suppresses creation of the assembly-listing and cross-reference files. A semicolon after the object-file name suppresses just the listing and cross-reference files. A semicolon after the listing-file name suppresses only the cross-reference file.

All files created during the assembly are written to the current drive and directory unless you specify a different drive for each file. You must separately specify the alternate drive and path for each file that you do not want to go on the current directory.

You can also specify a device name instead of a file name—for example, **NUL** for no file or **PRN** for the printer.

Note

If you want the file name for a given file to be the default (the file name of the source file), place the commas that would otherwise separate the file name from the other names side by side (,,). Unless a semicolon (;) is used, all the commas in the command line are required.

Spaces in a command line are optional. If you make an error entering any of the file names, **MASM** displays an error message and prompts for new file names, using the method described in Section 2.1.2, "Assembly Using Prompts."

■ Examples

```
MASM file.asm, file.obj, file.lst, file.crf
```

The example above is equivalent to the command line below:

```
MASM file,,,;
```

The source file `file.asm` is assembled. The generated relocatable code is copied to the object file `file.obj`. **MASM** also creates an assembly listing and a cross-reference file. These are written to `file.lst` and `file.crf`, respectively.

```
MASM startup,,stest;
```

The example above directs **MASM** to assemble the source file `startup.asm`. The assembler then writes the relocatable object code to the default object file, `startup.obj`. **MASM** creates a listing file named `stest.lst`, but the semicolon keeps the assembler from creating a cross-reference file.

```
MASM startup,,stest,;
```

The example above is the same as the previous example except that the semicolon follows a comma that marks the place of the cross-reference file. The assembler creates a cross-reference file `startup.crf`.

```
MASM  B:\src\build;
```

The example above directs **MASM** to find and assemble the source file `build.asm` in the directory `\src` on Drive B. The semicolon causes the assembler to create an object file named `build.obj` in the current directory, but prevents **MASM** from creating an assembly-listing or cross-reference file. Note that the object file is placed on the current drive, not the drive specified for the source file.

2.1.2 Assembly Using Prompts

You can direct **MASM** to prompt you for the files it needs by starting **MASM** with just the command name. **MASM** prompts you for the input it needs by displaying the following lines, one at a time:

```
Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:
```

The prompts correspond to the fields of **MASM** command lines. **MASM** waits for you to respond to each prompt before printing the next one. You must type a source-file name (though the extension is optional) at the first prompt. For other prompts, you can either type a file name, or press the ENTER key to accept the default displayed in brackets after the prompt.

File names typed at prompts must follow the command-line rules described in Section 2.1.1, "Assembly Using a Command Line." You can type options after any of the prompts as long as you separate them from file names with spaces. At any prompt, you can type the rest of the file names in the command-line format. For example, you can choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt (as long as you have supplied a source-file name), or you can type commas (,) to indicate several files.

After you have answered the last prompt and pressed the ENTER key, **MASM** assembles the source file.

2.2 Using Environment Variables

The Macro Assembler recognizes two environment variables: **INCLUDE** and **MASM**. The subsections below describe these environment variables and their use with the assembler.

Environment variables are described in general in the DOS user's guide.

2.2.1 The INCLUDE Environment Variable

The **INCLUDE** environment variable can specify the directory where include files are stored. This makes maintenance of include files easier, particularly on a hard disk. All include files can be kept in the same directory. If you keep source files in different directories, you do not have to keep copies of include files in each directory.

The **INCLUDE** environment variable is used by **MASM** only if you give a file name as an argument to the **INCLUDE** directive (see Section 11.6.1, "Using Include Files") If you give a complete file specification, including directory or drive, **MASM** only looks for the file in the specified directory.

When a file name is specified, **MASM** looks for the include file first in any directory specified with the **/I** option (see Section 2.4.6, "Getting Command-Line Help") If the **/I** option is not used or if the file is not found, **MASM** next looks in the current directory. If the file is still not

found, **MASM** looks in the directories specified with the **INCLUDE** environment variable in the order specified.

■ Examples

```
SET INCLUDE=C:\INCLUDE
```

This line defines the **INCLUDE** environment string to be `C:\INCLUDE`. Include files placed in this directory can be found automatically by **MASM**. You can put this line in your **AUTOEXEC.BAT** file to set the environment string each time you turn on your computer.

2.2.2 The MASM Environment Variable

The **MASM** environment variable can be used to specify default assembler options. If you define the options you use most in the environment variable, you do not need to type them on the command line every time you start the Macro Assembler.

When you start **MASM**, it reads the options in the environment variable first. Then it reads the options in the command line. If conflicting options are encountered, the last one read takes effect. This means that you can override default options in the environment variable by giving conflicting options in the command line.

Some options define the default action. If given by themselves, they have no effect since the default action is taken anyway. However, they are useful for overriding a nondefault action specified by an option in the environment variable.

Some assembler directives have the same effect as options. They always override related options.

Note

The equal sign (=) is not allowed in environment variables. Therefore the **/D** option when used with the equal sign cannot be put in an environment variable. For example, the following DOS command line is illegal and will cause a syntax error:

```
SET MASM=/Dtest=5
```

■ Examples

```
SET MASM=/A/ZI/Z
```

The command line above sets the **MASM** environment variable so that the **/A**, **/ZI**, and **/Z** options are in effect. The line can be put in an **AUTOEXEC.BAT** file to automatically set these options each time you start your computer.

Assume you have set the **MASM** environment string using the line shown above, and you then start **MASM** with the following command line:

```
MASM /S test;
```

The **/S** option, which specifies sequential segment ordering, conflicts with the **/A** option, which specifies alphabetical segment ordering. The command-line option overrides the environment option, and the source file has sequential ordering. (See Section 5.2.1, "Setting the Segment-Order Method," for information on the significance of segment order.)

However, if the source file contains the **.ALPHA** directive, it overrides all options and specifies alphabetical segment order.

2.3 Controlling Message Output

During and immediately after assembly, **MASM** sends messages to the standard output device. By default, this device is the screen. However, the display can be redirected so that instead it goes to a file or to a device such as a printer.

The messages can include a status message for successful assembly and error messages for unsuccessful assembly. The message format and the error and warning messages are described in Appendix B, "Error Messages and Exit Codes."

Some text-editing programs can use error information to locate errors in the source file. Typically, **MASM** is run as a shell from the editor and the assembler output is redirected into a file. The editor then opens the file and uses the data in it to locate errors in the source code. The errors may be located by line number, or by a search for the text of the error line.

If your text editor does not support this capability directly, you may still be able to use keystroke macros to set up similar functions. This requires either an editor that supports keystroke macros or a keyboard enhancer such as ProKey® or SuperKey®.

■ **Example**

```
MASM file; > errors
```

This command line sends to the file `errors` all messages that would normally be sent to the screen.

2.4 Using MASM Options

The **MASM** options control the operation of the assembler and the format of the output files it generates. Options can be entered with any combination of uppercase and lowercase letters.

MASM has the following options:

Option	Action
/A	Writes segments in alphabetical order
/B*number*	Sets buffer size
/C	Specifies a cross-reference file
/D	Creates Pass 1 listing
/D*symbol*[=*value*]	Defines assembler symbol
/E	Creates code for emulated floating-point instructions
/H	Lists command-line syntax and all assembler options
/I*path*	Sets include-file search path
/L	Specifies an assembly-listing file
/ML	Makes names case sensitive
/MU	Converts names to uppercase letters
/MX	Makes public and external names case sensitive
/N	Suppresses tables in listing file
/P	Checks for impure code
/S	Writes segments in source-code order
/T	Suppresses messages for successful assembly
/V	Displays extra statistics to screen
/W{0 | 1 | 2}	Sets error-display level

/X	Includes false conditionals in listings
/Z	Displays error lines on screen
/ZD	Puts line-number information in the object file
/ZI	Puts symbolic and line-number information in the object file

Note

Previous versions of the assembler provided a **/R** option to enable 8087 instructions and real numbers in the IEEE format. Since the current version of the assembler enables 8087 instructions and IEEE format by default, the **/R** option is no longer needed. The option is still recognized so that old make and batch files will work, but it has no effect. The previous default format, Microsoft Binary, can be specified with the **.MSFLOAT** directive, as described in Section 4.4.1, "Defining Default Assembly Behavior."

2.4.1 Specifying the Segment-Order Method

- **Syntax**

/S Default
/A

The **/A** option directs **MASM** to place the assembled segments in alphabetical order before copying them to the object file. The **/S** option directs the assembler to write segments in the order in which they appear in the source code.

Source-code order is the default. If no option is given, **MASM** copies the segments in the order encountered in the source file. The **/S** option is provided for compatibility with the XENIX® operating system and for overriding a default option in the **MASM** environment variable.

Note

Some previous versions of the IBM Macro Assembler ordered segments alphabetically by default. Listings in some books and magazines have been written with these early versions in mind. If you have trouble assembling and linking a listing taken from a book or magazine, try using the **/A** option.

The order in which segments are written to the object file is only one factor in determining the order in which they will appear in the executable file. The significance of segment order and ways to control it are discussed in Sections 5.2.1, "Setting the Segment-Order Method" and 5.2.2.3, "Defining Segment Combinations with Combine Type."

■ Example

```
MASM /A file;
```

The example above creates an object file, `FILE.OBJ`, whose segments are arranged in alphabetical order. If the **/S** option were used instead, or if no option were specified, the segments would be arranged in sequential order.

2.4.2 Setting the File-Buffer Size

■ Syntax

/Bnumber

The **/B** option directs the assembler to change the size of the file buffer used for the source file. The *number* is the number of 1024-byte (1K) memory blocks allocated for the buffer. You can set the buffer to any size from 1K to 63K (but not 64K). The default size of the buffer is 32K.

A buffer larger than your source file allows you to do the entire assembly in memory, greatly increasing assembly speed. However, you may not be able to use a large buffer if your computer does not have enough memory or if you have too many resident programs using up memory. If you get an error message indicating insufficient memory, you can decrease the buffer size and try again.

■ Examples

```
MASM /B16 file;
```

The example above decreases the buffer size to 16K.

```
MASM /B63 file;
```

The example above increases the buffer size to 63K.

2.4.3 Creating a Pass 1 Listing

■ Syntax

/D

The **/D** option tells **MASM** to add a Pass 1 listing to the assembly-listing file, making the assembly listing show the results of both assembler passes. A Pass 1 listing is typically used to locate phase errors. Phase errors occur when the assembler makes assumptions about the program in Pass 1 that are not valid in Pass 2.

The **/D** option does not create a Pass 1 listing unless you also direct **MASM** to create an assembly listing. It does direct the assembler to display error messages for both Pass 1 and Pass 2 of the assembly, even if no assembly listing is created. See Section 2.5.7 for more information about Pass 1 listings.

■ Example

```
MASM /D file,,;
```

This example directs the assembler to create a Pass 1 listing for the source file `file.asm`. The file `file.lst` will contain both the first and second pass listings.

2.4.4 Defining Assembler Symbols

■ Syntax

/D*symbol*[= *value*]

The **/D** option when given with a *symbol* argument directs **MASM** to define a symbol that can be used during the assembly as if it were defined as a text equate in the source file. Multiple symbols can be defined in a single command line.

The *value* can be any text string that does not include a space, comma, or semicolon. If no *value* is given, the symbol is assigned a null string.

As noted in Section 2.2.2, "The MASM Environment Variable," the version of the option using the equal sign cannot be stored in the **MASM** environment variable.

■ **Example**

```
MASM /Dwide /Dmode=3 file,,;
```

This example defines the symbol `wide` and gives it a null value. The symbol could then be used in the following conditional-assembly block:

```
IFDEF wide
PAGE 50,132
ENDIF
```

When the symbol is defined in the command line, the listing file is formatted for a 132-column printer. When the symbol is not defined in the command line, the listing file is given the default width of 80 (see the description of the **PAGE** directive in Section 12.2, "Controlling Page Format in Listings").

The example also defines the symbol `mode` and gives it the value 3. The symbol could then be used in a variety of contexts, as shown below:

```
            IF      mode LT 256         ; Use in expression
scrmode     DB      mode                ; Initialize byte variable
            ELSE
scrmode     DW      mode                ; Initialize word variable
            ENDIF
```

2.4.5 Creating Code for a Floating-Point Emulator

■ **Syntax**

/E

The **/E** option directs the assembler to generate data and code in the format expected by coprocessor emulator libraries. An emulator library uses 8088/8086 instructions to emulate the instructions of the 8087, 80287, or 80387 coprocessors. An emulator library can be used if you want your code to take advantage of a math coprocessor, or an emulator library can be used if the machine does not have a coprocessor.

Emulator libraries are only available with high-level-language compilers, including the Microsoft C, BASIC, FORTRAN, and Pascal compilers. The option cannot be used in stand-alone assembler programs unless you write your own emulator library. You cannot simply link with the emulator library from a high-level language, since these libraries require that the compiler start-up code be executed.

The Microsoft high-level-language compilers allow you to use options to specify whether you want to use emulator code. If you link a high-level-language module prepared with emulator options with an assembler module that uses coprocessor instructions, you should use the **/E** option when assembling.

To the applications programmer, writing code for the emulator is like writing code for a coprocessor. The instruction sets are the same (except as noted in Chapter 19, "Calculating with a Math Coprocessor"). However, at run time the coprocessor instructions are used only if there is a coprocessor available on the machine. If there is no coprocessor, the slower code from the emulator library is used instead.

■ Example

```
MASM /E /MX math.asm;
CL /FPi calc.c math
```

In the first command line, the source file `math.asm` is assembled with **MASM** by using the **/E** option. Then the **CL** program of the C compiler is used to compile the C source file `calc.c` with the **/FPi** option and finally to link the resulting object file (`calc.obj`) with `math.obj`. The compiler generates emulator code for floating-point instructions. There are similar options for the FORTRAN, BASIC, and Pascal compilers.

2.4.6 Getting Command-Line Help

■ Syntax

/H

The **/H** displays the command-line syntax and all the **MASM** options on the screen. You should not give any file names or other options with the **/H** option.

■ Example

```
MASM /H
```

2.4.7 Setting a Search Path for Include Files

■ Syntax

/Ipath

The **/I** option is used to set search paths for include files. You can set as many as 10 search paths by using the option for each path. The order of searching is the order in which the paths are listed in the command line. The **INCLUDE** directive and include files are discussed in Section 11.6.1, "Using Include Files."

■ Example

```
MASM /Ib:\io /I\macro file;
```

This command line might be used if the source file contains the following statement:

```
INCLUDE dos.inc
```

In this case, **MASM** would search for the file `dos.inc` first in directory `\io` on Drive B, and then in directory `\macro` on the current drive. If the file was not found in either of these directories, **MASM** would look next in the current directory and finally in any directories specified with the **INCLUDE** environment variable.

You should not specify a path name with the **INCLUDE** directive if you plan to specify search paths from the command line. For example, **MASM** would only search path `a:\macro` and would ignore any search paths specified in the command line if the source file contained any of the following statements:

```
INCLUDE a:\macro\dos.inc
INCLUDE ..\dos.inc
INCLUDE .\dos.inc
```

2.4.8 Specifying Listing and Cross-Reference Files

■ Syntax

/L
/C

The **/L** option directs **MASM** to create a listing file even if one was not specified in the command line or in response to prompts. The **/C** option

has the same effect for cross-reference files. Files specified with these options always have the base name of the source file plus the extension **.LST** for listing files or **.CRF** for cross-reference files. You cannot specify any other file name. Both options are provided for compatibility with the XENIX operating system.

■ Example

```
MASM /L /C file;
```

This line creates `file.lst` and `file.crf`. It is equivalent to the following command line:

```
MASM file,,,;
```

2.4.9 Specifying Case Sensitivity

■ Syntax

/MU Default
/ML
/MX

The **/ML** option directs the assembler to make all names case sensitive. The **/MX** option directs the assembler to make public and external names case sensitive. The **/MU** option directs the assembler to convert all names into uppercase letters.

By default, **MASM** converts all names into uppercase letters. The **/MU** option is provided for compatibility with XENIX (which uses **/ML** by default) and to override options given in the environment variable.

If case sensitivity is turned on, all names that have the same spelling, but use letters of different cases, are considered different. For example, with the **/ML** option, `DATA` and `data` are different. They would also be different with the **/MX** option if they were declared external or public. Public and external names include any label, variable, or symbol names defined by using the **EXTRN**, **PUBLIC**, or **COMM** directives (see Chapter 8, "Creating Programs from Multiple Modules").

If you use the **/ZI** or **/ZD** option (see Section 2.4.14, "Listing False Conditionals"), the **/MX**, **/ML**, and **/MU** options affect the case of the symbolic data that will be available to a symbolic debugger.

The /ML and /MX options are typically used when object modules created with MASM are to be linked with object modules created by a case-sensitive compiler such as the Microsoft C compiler. If case sensitivity is important, you should also use the linker /NOI option.

■ **Example**

```
MASM /MX module;
LINK /NOI module;
```

This example shows how to use the /MX option with MASM to assemble a file with case-sensitive public symbols.

2.4.10 Suppressing Tables in the Listing File

■ **Syntax**

/N

The /N option tells the assembler to omit all tables from the end of the listing file. If this option is not chosen, MASM includes tables of macros, structures, records, segments and groups, and symbols. The code portion of the listing file is not changed by the /N option.

■ **Example**

```
MASM /N file,,;
```

2.4.11 Checking for Impure Code

■ **Syntax**

/P

The /P option directs MASM to check for impure code in the 80286 or 80386 privileged mode.

Code that moves data into memory with a CS: override is acceptable in real mode. However, such code may cause problems in protected mode. When the /P option is in effect, the assembler checks for these situations and generates an error if it encounters them.

Real and privileged modes are explained in Chapter 13, "Understanding 8086-Family Processors." Versions of DOS available at release time do not support privileged mode.

This option is provided for XENIX compatibility and to warn about programming practices that will be illegal under OS/2, the planned multitasking operating system.

■ **Example**

```
                .CODE
                .
                .
                .
                jmp     past            ; Don't execute data
addr            DW      ?               ; Allocate code space for data
past:           .
                .                       ; Calculate value of "addr" here
                .
                mov     cs:addr,si      ; Load register address
```

The example shows a **CS** override. If assembled with the **/P** option, an error is generated.

2.4.12 Controlling Display of Assembly Statistics

■ **Syntax**

/V
/T

The **/V** and **/T** options specify the level of information displayed to the screen at the end of assembly. (**V** is a mnemonic for verbose; **T** is a mnemonic for terse.)

If neither option is given, **MASM** outputs a line telling the amount of symbol space free and the number of warnings and errors.

If the **/V** option is given, **MASM** also reports the number of lines and symbols processed.

If the **/T** option is given, **MASM** does not output anything to the screen unless errors are encountered. This option may be useful in batch or make files if you do not want the output cluttered with unnecessary messages.

If errors are encountered, they will be displayed whether these options are given or not. Appendix B, "Error Messages and Exit Codes," describes the messages displayed after assembly.

2.4.13 Setting the Warning Level

- **Syntax**

/W{0 | 1 | 2}

The **/W** option sets the assembler warning level. **MASM** gives warning messages for assembly statements that are ambiguous or questionable but not necessarily illegal. Some programmers purposely use practices that generate warnings. By setting the appropriate warning level, they can turn off warnings if they are aware of the problem and do not wish to take action to remedy it.

MASM has three levels of errors, as shown in Table 2.1.

Table 2.1
Warning Levels

Level	Type	Description
0	Severe errors	Illegal statements
1	Serious warnings	Ambiguous statements or questionable programming practices
2	Advisory warnings	Statements that may produce inefficient code

The default warning level is 1. A higher warning level includes a lower level. Level 2 includes severe errors, serious warnings, and advisory warnings. If severe errors are encountered, no object file is produced.

The advisory warnings are listed below:

Number	Message
104	`Operand size does not match word size`

105	Address size does not match word size
106	Jump within short distance

The serious warnings are listed below:

Number	Message
1	Extra characters on line
16	Symbol is reserved word
31	Operand types must match
57	Illegal size for item
85	End of file, no END directive
101	Missing data; zero assumed
102	Segment near (or at) 64k limit

All other errors are severe.

2.4.14 Listing False Conditionals

■ Syntax

/X

The **/X** option directs **MASM** to copy to the assembly listing all statements forming the body of conditional-assembly blocks whose condition is false. If you do not give the **/X** option in the command line, **MASM** suppresses all such statements. The **/X** option lets you display conditionals that do not generate code. Conditional-assembly directives are explained in Chapter 12, "Controlling Assembly Output."

The **.LFCOND**, **.SFCOND**, and **.TFCOND** directives can override the effect of the **/X** option, as described in Section 12.3.2, "Controlling Listing of Conditional Blocks." The **/X** option does not affect the assembly listing unless you direct the assembler to create an assembly-listing file.

■ Example

```
MASM /X file,,;
```

Listing of false conditionals is turned on when `file.asm` is assembled. Directives in the source file can override the **/X** option to change the status of false-conditional listing.

2.4.15 Displaying Error Lines on the Screen

■ **Syntax**

/Z

The **/Z** option directs **MASM** to display lines containing errors on the screen. Normally when the assembler encounters an error, it displays only an error message describing the problem. When you use the **/Z** option in the command line, the assembler displays the source line that produced the error in addition to the error message. **MASM** assembles faster without the **/Z** option, but you may find the convenience of seeing the incorrect source lines worth the slight cost in processing speed.

■ **Example**

```
MASM /Z file;
```

2.4.16 Writing Symbolic Information to the Object File

■ **Syntax**

/ZI
/ZD

The **/ZI** option directs **MASM** to write symbolic information to the object file. There are two types of symbolic information available: line-number data and symbolic data.

Line-number data relates each instruction to the source line that created it. The CodeView debugger and **SYMDEB** (the debugger provided with some earlier versions of **MASM**) need this information for source-level debugging.

Symbolic data specifies a size for each variable or label used in the program. This includes both public and nonpublic labels and variable names. Public symbols are discussed in Chapter 8, "Creating Programs from Multiple Modules." The CodeView debugger (but not **SYMDEB**) uses this information to specify the correct size for data objects so that they can be used in expressions.

The **/ZI** option writes both line-number and symbolic data to the object file. If you plan to debug your programs with the CodeView debugger, use the **/ZI** option when assembling and the **/CO** option when linking. All the necessary debugging information is available in executable files prepared in the **.EXE** format. Debugging information is stripped out of programs prepared in **.COM** format.

The **/ZD** option writes line-number information only to the object file. It can be used if you plan to debug with **SYMDEB** or if you want to see line numbers in map files. The **/ZI** option can also be used for these purposes, but it produces larger object files. If you do not have enough memory to debug a program with the CodeView debugger, you can reduce the program size by using **/ZD** instead of **/ZI** for all or some modules.

The option names **/ZI** and **/ZD** are similar to corresponding option names for recent versions of Microsoft compilers.

2.5 Reading Assembly Listings

MASM creates an assembly listing of your source file whenever you give an assembly-listing file name on the **MASM** command line or in response to the **MASM** prompts. The assembly listing contains both the statements in the source file and the object code (if any) generated for each statement. The listing also shows the names and values of all labels, variables, and symbols in your source file.

The assembler creates tables for macros, structures, records, segments, groups, and other symbols. These tables are placed at the end of the assembly listing (unless you suppress them with the **/N** option). **MASM** lists only the types of symbols encountered in the program. For example, if your program has no macros, there will be no macro section in the symbol table. All symbol names will be shown in uppercase letters unless you use the **/ML** or **/MX** option to specify case sensitivity.

2.5.1 Reading Code in a Listing

The assembler lists the code generated from the statements of a source file. Each line has the syntax shown below:

[[linenumber]] offset [[code]] statement

The *linenumber* is the number of the line starting from the first statement in the assembly listing. Line numbers are produced only if you request a

cross-reference file. Line numbers in the listing do not always correspond to the same lines in the source file.

The *offset* is the offset from the beginning of the current segment to the code. If the statement generates code or data, *code* shows the numeric value in hexadecimal if the value is known at assembly time. If the value is calculated at run time, **MASM** indicates what action is necessary to compute the value. The *statement* is the source statement shown exactly as it appears in the source file, or as expanded by a macro.

If any errors occur during assembly, each error message and error number will appear directly below the statement where the error occurred. Refer to Appendix B, "Error Messages and Exit Codes," for a list of **MASM** errors and a discussion of the format in which errors are displayed. An example of an error line and message is shown below:

```
     71 0012  E8 001C R                                call    doit
test.ASM(46): error A2071: Forward needs override or FAR
```

Note that number 46 in the error message is the source line where the error occurred. Number 71 on the code line is the listing line where the error occurred. These lines will seldom be the same.

The assembler uses the symbols and abbreviations in Table 2.2 to indicate addresses that need to be resolved by the linker or values that were generated in a special way.

Table 2.2
Symbols and Abbreviations in Listings

Character	Meaning
R	Relocatable address (linker must resolve)
E	External address (linker must resolve)
----	Segment/group address (linker must resolve)
=	**EQU** or equal-sign (=) directive
nn:	Segment override in statement
nn/	**REP** or **LOCK** prefix instruction
nn[*xx*]	**DUP** expression: *nn* copies of the value *xx*
n	Macro-expansion nesting level (+ if more than nine)
C	Line from **INCLUDE** file
|	80386 size or address prefix

■ Example

The sample listing shown in this section is produced by using the **/ZI** option. A cross-reference file is specified so that line numbers will appear in the listing. The command line is as follows:

```
MASM /ZI listdemo,,,;
```

The code portion of the resulting listing is shown below. The tables normally seen at the end of the listing are explained later, in Sections 2.5.2–2.5.7 below.

```
Microsoft (R) Macro Assembler Version 5.00                    9/22/87 14:44:53
Listing features demo                                         Page     1-1


                                                PAGE    65,132
                                                TITLE   Listing features demo
                                      C         INCLUDE dos.mac
                                      C  StrAlloc    MACRO   name,text
                                      C  name        DB      &text
                                      C              DB      13d,10d
                                      C  l&name      EQU     $-name
                                      C              ENDM


= 0080                                  larg    EQU     80h

                                                DOSSEG
                                                .MODEL  small

0100                                            .STACK  256

                                        color   RECORD  b:1,r:3=1,i:1=1,f:3=7

                                        date    STRUC
0000  05                                month   DB      5
0001  07                                day     DB      7
0002  07C3                              year    DW      1987
0004                                    date    ENDS

0000                                            .DATA
0000  1F                                text    color   <>
0001  09                                today   date    <9,22,1987>
0002  16
0003  07C3

0005  0064[                             buffer  DW      100 DUP(?)
            ????
                                ]


                                                StrAlloc ending,"Finished."
00CD  46 69 6E 69 73 68 65  1  ending          DB      "Finished."
00D6  0D 0A                 1                  DB      13d,10d

```

```
      41 0000                                        .CODE
      42
      43 0000  B8 ---- R                    start:   mov     ax,@DATA
      44 0003  8E D8                                 mov     ds,ax
      45
      46 0005  B8 0063                               mov     ax,'c'
      47 0008  26: 8B 0E 0080                        mov     cx,es:larg
      48 000D  BF 0052                               mov     di,82
      49 0010  F2/ AE                                repne   scasb
      50 0012  57                                    push    di
      51
      52                                             EXTRN   work:NEAR
      53 0013  E8 0000 E                             call    work
      54
      55 0016  B8 170C                               mov     ax,4C00
listdemo.ASM(40): error A2107: Non-digit in number
      56 0019  CD 21                                 int     21h
      57
      58 001B                                        END     start
```

2.5.2 Reading a Macro Table

A macro table at a listing file's end gives in alphabetical order the names and sizes (in lines) of all macros called or defined in the source file.

■ Example

```
Macros:

                N a m e                 Lines

STRALLOC . . . . . . . . . . . . .        3
```

2.5.3 Reading a Structure and Record Table

All structures and records declared in the source file are given at the end of the listing file. The names are listed in alphabetical order. Each name is followed by the fields in the order in which they are declared.

■ Example

```
Structures and Records:

                N a m e                 Width   # fields
                                        Shift   Width   Mask    Initial

COLOR  . . . . . . . . . . . . . .      0008    0004
  B    . . . . . . . . . . . . . .      0007    0001    0080    0000
  R    . . . . . . . . . . . . . .      0004    0003    0070    0010
  I    . . . . . . . . . . . . . .      0003    0001    0008    0008
  F    . . . . . . . . . . . . . .      0000    0003    0007    0007
DATE . . . . . . . . . . . . . . .      0004    0003
  MONTH  . . . . . . . . . . . . .      0000
  DAY  . . . . . . . . . . . . . .      0001
  YEAR . . . . . . . . . . . . . .      0002
```

The first row of headings only applies to the record or structure itself. For a record, the "Width" column shows the width in bits while the "# fields" column tells the total number of fields.

The second row of headings applies only to fields of the record or structure. For records, the "Shift" column lists the offset (in bits) from the low-order bit of the record to the low-order bit in the field. The "Width" column lists the number of bits in the field. The "Mask" column lists the maximum value of the field, expressed in hexadecimal. The "Initial" column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0.

For a structure, the "Width" column lists the size of the structure in bytes. The "# fields" column lists the number of fields in the structure. Both values are in hexadecimal.

For structure fields, the "Shift" column lists the offset in bytes from the beginning of the structure to the field. This value is in hexadecimal. The other columns are not used.

2.5.4 Reading a Segment and Group Table

Segments and groups used in the source file are listed at the end of the program with their size, align type, combine type, and class. If you used simplified segment directives in the source file, the actual segment names generated by **MASM** will be listed in the table.

■ Example

```
Segments and Groups:

                N a m e                 Size    Align   Combine Class

DGROUP . . . . . . . . . . . . . .      GROUP
  _DATA  . . . . . . . . . . . . .      00D8    WORD    PUBLIC  'DATA'
  STACK  . . . . . . . . . . . . .      0800    PARA    STACK   'STACK'
_TEXT  . . . . . . . . . . . . . .      0018    BYTE    PUBLIC  'CODE'
```

The "Name" column lists the names of all segments and groups. Segment and group names are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name in the order in which they were added to the group.

The "Size" column lists the byte size (in hexadecimal) of each segment. The size of groups is not shown.

The "Align" column lists the align type of the segment.

The "Combine" column lists the combine type of the segment. If no explicit combine type is defined for the segment, the listing shows NONE, representing the private combine type. If the "Align" column contains AT, the "Combine" column contains the hexadecimal address of the beginning of the segment.

The "Class" column lists the class name of the segment. For a complete explanation of the align, combine, and class types, see Section 5.2.2, "Defining Full Segments."

2.5.5 Reading a Symbol Table

All symbols (except names for macros, structures, records, and segments) are listed in a symbol table at the end of the listing.

■ Example

```
Symbols:

                N a m e                 Type     Value   Attr

BUFFER . . . . . . . . . . . . .        L WORD   0005    _DATA   Length = 0064

ENDING . . . . . . . . . . . . .        L BYTE   00CD    _DATA

LARG . . . . . . . . . . . . . .        NUMBER   0080
LENDING  . . . . . . . . . . . .        NUMBER   000B

START  . . . . . . . . . . . . .        L NEAR   0000    _TEXT

TEXT . . . . . . . . . . . . . .        L BYTE   0000    _DATA
TODAY  . . . . . . . . . . . . .        L DWORD  0001    _DATA

WORK . . . . . . . . . . . . . .        L NEAR   0000    _TEXT   External

@CODE  . . . . . . . . . . . . .        TEXT  _TEXT
@CODESIZE  . . . . . . . . . . .        TEXT  0
@DATA  . . . . . . . . . . . . .        TEXT  DGROUP
@DATASIZE  . . . . . . . . . . .        TEXT  0
@FARDATA . . . . . . . . . . . .        TEXT  FAR_DATA
@FARDATA?  . . . . . . . . . . .        TEXT  FAR_BSSk
@FILENAME  . . . . . . . . . . .        TEXT  listdemo
```

The "Name" column lists the names in alphabetical order. The "Type" column lists each symbol's type. A type is given as one of the following:

Type	Definition
L NEAR	A near label
L FAR	A far label
N PROC	A near procedure label
F PROC	A far procedure label
NUMBER	An absolute label
ALIAS	An alias for another symbol
OPCODE	An equate for an instruction opcode
TEXT	A text equate
BYTE	One byte
WORD	One word (two bytes)
DWORD	Doubleword (four bytes)
FWORD	Farword (six bytes)
QWORD	Quadword (eight bytes)
TBYTE	Ten bytes
number	Length in bytes of a structure variable

The length of a multiple-element variable such as an array or string is the length of a single element, not the length of the entire variable. For example, string variables are always shown as **L BYTE**.

If the symbol represents an absolute value defined with an **EQU** or equal-sign (=) directive, the "Value" column shows the symbol's value. The value may be another symbol, a string, or a constant numeric value (in hexadecimal), depending on whether the type is **ALIAS**, **TEXT**, or **NUMBER**. If the type is **OPCODE**, the "Value" column will be blank. If the symbol represents a variable, label, or procedure, the "Value" column shows the symbol's hexadecimal offset from the beginning of the segment in which it is defined.

The "Attr" column shows the attributes of the symbol. The attributes include the name of the segment (if any) in which the symbol is defined, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the **EXTRN** and **PUBLIC** directives. The scope can be **EXTERNAL**, **GLOBAL**, or **COMMUNAL**. The code length (in hexadecimal) is given only for procedures. The "Attr" column is blank if the symbol has no attribute.

The text equates shown at the end of the sample table are the ones defined automatically when you use simplified segment directives (see Section 5.1.1, "Understanding Memory Models").

2.5.6 Reading Assembly Statistics

Data on the assembly, including the number of lines and symbols processed and the errors or warnings encountered, are shown at the end of the listing. See Appendix B, "Error Messages and Exit Codes," for further information on this data.

■ Example

```
    48 Source  Lines
    52 Total   Lines
    53 Symbols

 45570 + 310654 Bytes symbol space free

     0 Warning Errors
     1 Severe  Errors
```

2.5.7 Reading a Pass 1 Listing

When you specify the **/D** option in the **MASM** command line, the assembler puts a Pass 1 listing in the assembly-listing file. The listing file shows the results of both assembler passes. Pass 1 listings are useful in analyzing phase errors.

The following example illustrates a Pass 1 listing for a source file that assembled without error on the second pass.

```
 0017  7E 00                jle     label1
PASS_CMP.ASM(20) : error 9 : Symbol not defined LABEL1
 0019  BB 1000              mov     bx,4096
 001C              label1:
```

During Pass 1, the **JLE** instruction to a forward reference produces an error message, and the value 0 is encoded as the operand. **MASM** displays this error because it has not yet encountered the symbol `label1`.

Later in Pass 1, `label1` is defined. Therefore, the assembler knows about `label1` on Pass 2 and can fix the Pass 1 error. The Pass 2 listing is shown below:

```
 0017  7E 03                jle     label1
 0019  BB 1000              mov     bx,4096
 001C              label1:
```

The operand for the **JLE** instruction is now coded as 3 instead of 0 to indicate that the distance of the jump to `label1` is three bytes.

Since **MASM** generated the same number of bytes for both passes, there was no error. Phase errors occur if the assembler makes an assumption on Pass 1 that it cannot change on Pass 2. If you get a phase error, you can examine the Pass 1 listing to see what assumptions the assembler made.

CHAPTER 3

USING CREF

3.1 Using CREF 53
3.1.1 Using a Command Line to Create a Cross-Reference Listing 53
3.1.2 Using Prompts to Create a Cross-Reference Listing 54
3.2 Reading Cross-Reference Listings 55

The Microsoft Cross-Reference Utility (**CREF**) creates a cross-reference listing of all symbols in an assembly-language program. A cross-reference listing is an alphabetical list of symbols in which each symbol is followed by a series of line numbers. The line numbers indicate the lines in the source program that contain a reference to the symbol.

CREF is intended for use as a debugging aid to speed up the search for symbols encountered during a debugging session. The cross-reference listing, together with the symbol table created by the assembler, can make debugging and correcting a program easier.

3.1 Using CREF

CREF creates a cross-reference listing for a program by converting a binary cross-reference file, produced by the assembler, into a readable ASCII file. You create the cross-reference file by supplying a cross-reference-file name when you invoke the assembler. See Section 2.1.1, "Assembly Using a Command Line," for more information on creating a binary cross-reference file. You create the cross-reference listing by invoking **CREF** and supplying the name of the cross-reference file.

3.1.1 Using a Command Line to Create a Cross-Reference Listing

To convert a binary cross-reference file created by **MASM** into an ASCII cross-reference listing, type **CREF** followed by the names of the files you want to process.

■ Syntax

CREF *crossreferencefile* [,*crossreferencelisting*] [;]

The *crossreferencefile* is the name of the cross-reference file created by **MASM**, and the *crossreferencelisting* is the name of the readable ASCII file you wish to create.

If you do not supply file-name extensions when you name the files, **CREF** automatically provides **.CRF** for the cross-reference file and **.REF** for the cross-reference-listing file. If you do not want these extensions, you must supply your own.

You can select a default file name for the listing file by typing a semicolon (;) immediately after *crossreferencefile*.

You can specify a directory or disk drive for either of the files. You can also name output devices such as CON (display console) and PRN (printer).

When **CREF** finishes creating the cross-reference-listing file, it displays the number of symbols processed.

Examples

```
CREF  test.crf,test.ref
```

The example above converts the cross-reference file `test.crf` to the cross-reference-listing file `test.ref`. It is equivalent to

```
CREF  test,test
```

or

```
CREF  test;
```

The following example directs the cross-reference listing to the screen. No file is created.

```
CREF  test,con
```

3.1.2 Using Prompts to Create a Cross-Reference Listing

You can direct **CREF** to prompt you for the files it needs by starting **CREF** with just the command name. **CREF** prompts you for the input it needs by displaying the following lines, one at a time:

```
Cross-Reference [.CRF]:
Listing [filename.REF]:
```

The prompts correspond to the fields of **CREF** command lines. **CREF** waits for you to respond to each prompt before printing the next one. You must type a cross-reference file name (though the extension is optional) at the first prompt. For the second prompt, you can either type a file name or press the ENTER key to accept the default displayed in brackets after the prompt.

After you have answered the last prompt and pressed the ENTER key, **CREF** reads the cross-reference file and creates the new listing. It also displays the number of symbols in the cross-reference file.

3.2 Reading Cross-Reference Listings

The cross-reference listing contains the name of each symbol defined in your program. Each name is followed by a list of line numbers representing the line or lines in the listing file in which a symbol is defined or used. Line numbers in which a symbol is defined are marked with a number sign (#).

Each page in the listing begins with the title of the program. The title is the name or string defined by the **TITLE** directive in the source file (see Section 12.2.1, "Setting the Listing Title").

■ Example

The next three code samples illustrate source, listings, and cross-reference files for a program. The source file `hello.asm` is shown below:

```
                TITLE   hello

                DOSSEG
                .MODEL  small

                .STACK  100h

                .DATA
                PUBLIC  message,lmessage
message         DB      "Hello, world."
lmessage        EQU     $ - message

                .CODE

start:          mov     ax,DGROUP
                mov     ds,ax

                EXTRN   display:NEAR
                call    display

                mov     ax,4C00h
                int     21h

                END     start
```

To assemble the program and create a cross-reference file, enter the following command line:

```
MASM hello,,,;
```

The listing file `hello.lst` produced by this assembly is shown below:

```
Microsoft (R) Macro Assembler Version 5.00                  9/22/87 15:39:48
hello                                                       Page     1-1


       1                                            TITLE   hello
       2
       3                                            DOSSEG
       4                                            .MODEL  small
       5
       6 0100                                       .STACK  100h
       7
       8 0000                                       .DATA
       9                                            PUBLIC  message,lmessage
      10 0000  48 65 6C 6C 6F 2C 20     message     DB      "Hello, world."
      11       77 6F 72 6C 64 2E
      12 = 000D                         lmessage    EQU     $ - message
      13
      14 0000                                       .CODE
      15
      16 0000  B8 ---- R                start:      mov     ax,DGROUP
      17 0003  8E D8                                mov     ds,ax
      18
      19                                            EXTRN   display:NEAR
      20 0005  E8 0000 E                            call    display
      21
      22 0008  B8 4C00                              mov     ax,4C00h
      23 000B  CD 21                                int     21h
      24
      25 000D                                       END     start

Microsoft (R) Macro Assembler Version 5.00                  9/22/87 15:39:48
hello                                                       Symbols-1


Segments and Groups:

                N a m e                 Length   Align   Combine Class

DGROUP . . . . . . . . . . . . .        GROUP
  _DATA  . . . . . . . . . . . .        000D     WORD    PUBLIC  'DATA'
  STACK  . . . . . . . . . . . .        0100     PARA    STACK   'STACK'
_TEXT  . . . . . . . . . . . . .        000D     BYTE    PUBLIC  'CODE'

Symbols:

                N a m e                 Type     Value   Attr

DISPLAY  . . . . . . . . . . . .        L NEAR   0000    _TEXT   External

LMESSAGE . . . . . . . . . . . .        NUMBER   000D            Global

MESSAGE  . . . . . . . . . . . .        L BYTE   0000    _DATA   Global

START  . . . . . . . . . . . . .        L NEAR   0000    _TEXT

@CODE  . . . . . . . . . . . . .        TEXT  _text
@CODESIZE  . . . . . . . . . . .        TEXT  0
@DATA  . . . . . . . . . . . . .        TEXT  dgroup
@DATASIZE  . . . . . . . . . . .        TEXT  0
@FARDATA . . . . . . . . . . . .        TEXT  far_data
@FARDATA?  . . . . . . . . . . .        TEXT  far_bss
@FILENAME  . . . . . . . . . . .        TEXT  hellod


     24 Source  Lines
     24 Total   Lines
     39 Symbols

  45994 + 314294 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

To create a cross-reference listing of the file `hello.crf`, enter the following command line:

```
CREF hello;
```

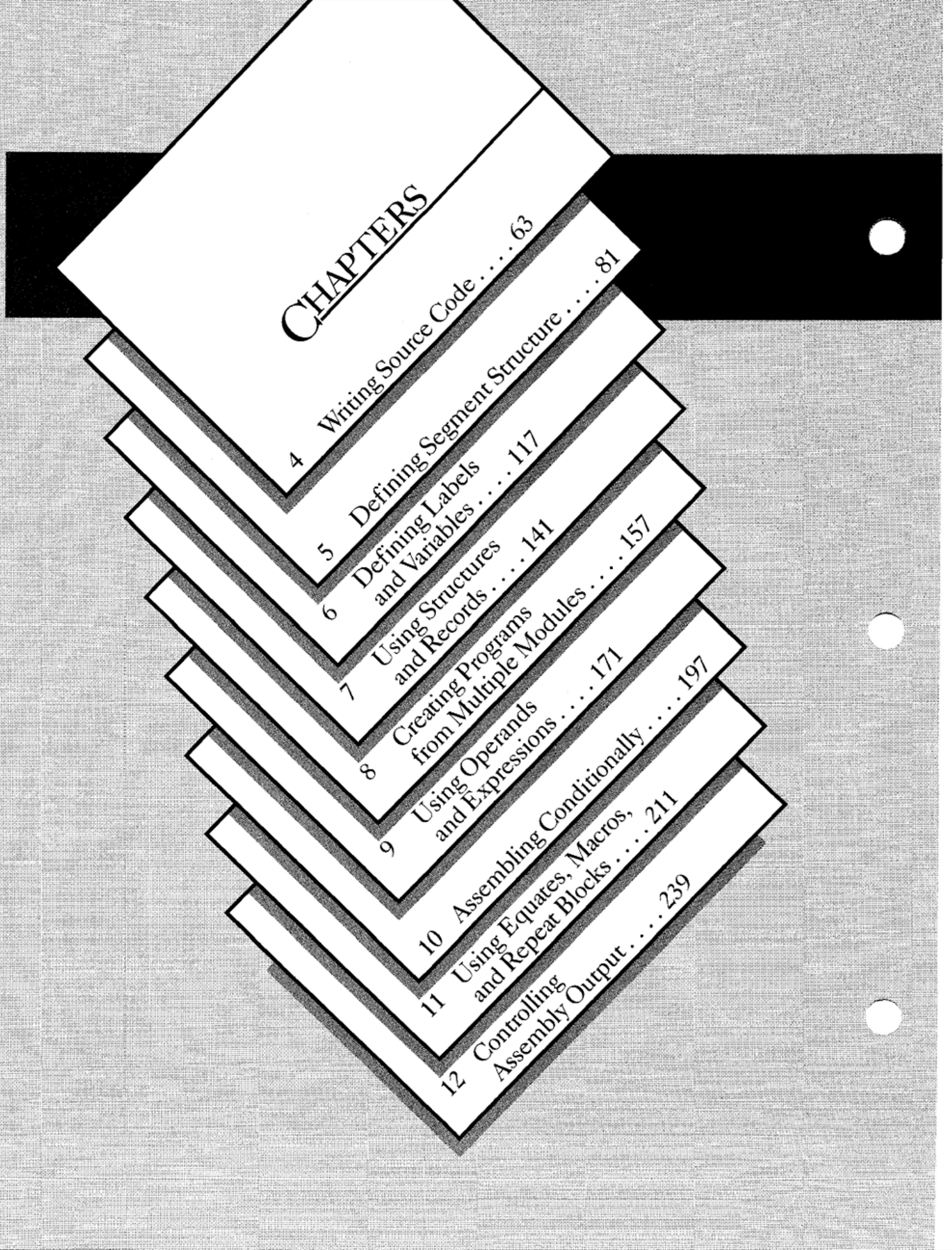
The resulting cross-reference-listing file `hello.ref` is shown below:

```
Microsoft Cross-Reference  Version 5.00                  9/22/87 15:39:48
hello

  Symbol Cross-Reference            (# is definition)              Cref-1

CODE . . . . . . . . . . . . . .   14

DATA . . . . . . . . . . . . . .    8
DGROUP . . . . . . . . . . . . .   16
DISPLAY. . . . . . . . . . . . .   19#    20

LMESSAGE . . . . . . . . . . . .    9     12#

MESSAGE. . . . . . . . . . . . .    9     10#    12

STACK. . . . . . . . . . . . . .    6#     6
START. . . . . . . . . . . . . .   16#    25

_DATA. . . . . . . . . . . . . .    8#
_TEXT. . . . . . . . . . . . . .   14#

 10 Symbols
```

Notice that line numbers in the listing and cross-reference-listing files may not identify corresponding lines in the source file.

PART 2

USING DIRECTIVES


CHAPTERS
4 Writing Source Code 63
5 Defining Segment Structure 81
6 Defining Labels and Variables 117
7 Using Structures and Records 141
8 Creating Programs from Multiple Modules 157
9 Using Operands and Expressions 171
10 Assembling Conditionally 197
11 Using Equates, Macros, and Repeat Blocks 211
12 Controlling Assembly Output 239

PART 2 ◇ USING DIRECTIVES

Part 2 of the Programmer's Guide (comprising chapters 4–12) describes the directives and operators recognized by the Microsoft Macro Assembler. Directives tell you how to generate code and data at assembly time. Operators tell you how to combine operands to form assembly-language expressions.

Chapter 4 introduces basic concepts of the assembly language recognized by the Microsoft Macro Assembler. Topics covered include symbols, constants, statement syntax, and processor directives.

Chapters 5–8 explain the different directives and operators. The material is organized topically, with related directives discussed together. Operators and expressions are discussed specifically in Chapter 9.

CHAPTER 4

WRITING SOURCE CODE

4.1 Writing Assembly-Language Statements 65
4.1.1 Using Mnemonics and Operands 66
4.1.2 Writing Comments 67
4.2 Assigning Names to Symbols 67
4.3 Constants 69
4.3.1 Integer Constants 70
4.3.1.1 Specifying Integers with Radix Specifiers 70
4.3.1.2 Setting the Default Radix 71
4.3.2 Packed Binary Coded Decimal Constants 72
4.3.3 Real-Number Constants 73
4.3.4 String Constants 74
4.4 Defining Default Assembly Behavior 75
4.5 Ending a Source File 78

Assembly-language programs are written as source files, which can then be assembled into object files by **MASM**. Object files can then be processed and combined with **LINK** to form executable files.

Source files are made up of assembly-language statements. Statements are in turn made up of mnemonics, operands, and comments. This chapter describes how to write assembly-language statements. Symbol names and constants are explained. It also tells you how to start and end assembly-language source files.

4.1 Writing Assembly-Language Statements

A statement is a combination of mnemonics, operands, and comments that defines the object code to be created at assembly time. Each line of source code consists of a single statement. Multiline statements are not allowed. Statements must not have more than 128 characters. Statements can have up to four fields, as shown below:

■ Syntax

[[*name*]] [[*operation*]] [[*operands*]] [[;*comment*]]

The fields are explained below, starting with the leftmost field:

Field	Purpose
name	Labels the statement so that the statement can be accessed by name in other statements
operation	Defines the action of the statement
operands	Defines the data to be operated on by the statement
comment	Describes the statement without having any effect on assembly

All fields are optional, although the operand or name fields may be required if certain directives or instructions are given in the operation field. A blank line is simply a statement in which all fields are blank. A comment line is a statement in which all fields except the comment are blank.

Statements can be entered in uppercase or lowercase letters. Sample code in this manual uses uppercase letters for directives, hexadecimal letter

digits, and segment definitions. Your code will be clearer if you choose a case convention and use it consistently.

Each field (except the comment field) must be separated from other fields by a space or tab character. That is the only limitation on structure imposed by **MASM**. For example, the following code is legal:

```
dosseg;use microsoft segment conventions
.model small;conventions and small model
.stack 100h;allocate 256-byte stack
.data
message db "Hello, world.",13,10;message to be written
lmessage equ $ - message;length of message
.code
start: mov ax,@data;load segment location
mov ds,ax;into ds register
mov bx,1;load 1 - file handle for
;standard output
mov cx,lmessage;load length of message
mov dx,offset message;load address of message
mov ah,40h;load number for dos write function
int 21h;call dos
mov ax,4c00h;load dos exit function (4ch)
;in ah and 0 errorlevel in al
int 21h;call dos
end start
```

However, the code is much easier to interpret if each field is assigned a specified tab position and a standard convention is used for capitalization. The example program in Chapter 1, "Getting Started," is the same as the example above except for the conventions used.

4.1.1 Using Mnemonics and Operands

Mnemonics are the names assigned to commands that tell either the assembler or the processor what to do. There are two types of mnemonics: directives and instructions.

Directives give directions to the assembler. They specify the manner in which the assembler is to generate object code at assembly time. Part 2, "Using Directives," describes the directives recognized by the assembler. Directives are also discussed in Part 3, "Using Instructions."

Instructions give directions to the processor. At assembly time, they are translated into object code. At run time, the object code controls the behavior of the processor. Instructions are described in Part 3, "Using Instructions."

Operands define the data that is used by directives and instructions. They can be made up of symbols, constants, expressions, and registers. Sections

4.2 and 4.3 below discuss symbol names and constants. Operands, expressions, and registers are discussed throughout the manual, but particularly in Chapter 9, "Using Operands and Expressions," and Chapter 14, "Using Addressing Modes."

4.1.2 Writing Comments

Comments are descriptions of the code. They are for documentation only and are ignored by the assembler.

Any text following a semicolon is considered a comment. Comments commonly start in the column assigned for the comment field, or in the first column of the source code. The comment must follow all other fields in the statement.

Multiline comments can either be specified with multiple comment statements or with the **COMMENT** directive.

- **Syntax**

COMMENT *delimiter* [[*text*]]
text
delimiter [[*text*]]

All *text* between the first *delimiter* and the line containing a second *delimiter* is ignored by the assembler. The *delimiter* character is the first nonblank character after the **COMMENT** directive. The *text* includes the comments up to and including the line containing the next occurrence of the delimiter.

- **Example**

```
                COMMENT + The plus
                        sign is the delimiter. The
                        assembler ignores the statement
                        following the last delimiter
+               mov     ax,1    (ignored)
```

4.2 Assigning Names to Symbols

A symbol is a name that represents a value. Symbols are one of the most important elements of assembly-language programs. Elements that must

be represented symbolically in assembly-language source code include variables, address labels, macros, segments, procedures, records, and structures. Constants, expressions, and strings can also be represented symbolically.

Symbol names are combinations of letters (both uppercase and lowercase), digits, and special characters. The Macro Assembler recognizes the following character set:

```
A-Z a-z 0-9
?  @  _  $ : . [ ] ( ) < > { } + - / *
& % ! ' ~ ¦ \ = # ^ ; , ` "
```

Letters, digits, and some characters can be used in symbol names, but some restrictions on how certain characters can be used or combined are listed below:

- A name can have any combination of uppercase and lowercase letters. All lowercase letters are converted to uppercase ones by the assembler, unless the **/ML** assembly option is used, or unless the name is declared with a **PUBLIC** or **EXTRN** directive and the **/MX** option is used.
- Digits may be used within a name, but not as the first character.
- A name can be given any number of characters, but only the first 31 are used. All other characters are ignored.
- The following characters may be used at the beginning of a name or within a name: underscore (_), question mark (?), dollar sign ($), and at sign (@).
- The period (.) is an operator and cannot be used within a name, but it can be used as the first character of a name.
- A name may not be the same as any reserved name. Note that two special characters, the question mark (?) and the dollar sign ($), are reserved names and therefore can't stand alone as symbol names.

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. All uppercase and lowercase letter combinations of these names are treated as the same name.

Table 4.1 lists names that are always reserved by the assembler. Using any of these names for a symbol results in an error.

Table 4.1

Reserved Names

$	.DATA	.ERRNDEF	.LALL	REPT
*	.DATA?	.ERRNZ	LE	.SALL
+	DB	EVEN	LENGTH	SEG
-	DD	EXITM	.LFCOND	SEGMENT
.	DF	EXTRN	.LIST	.SEQ
/	DOSSEG	FAR	LOCAL	.SFCOND
=	DQ	.FARDATA	LOW	SHL
?	DS	.FARDATA?	LT	SHORT
[ ]	DT	FWORD	MACRO	SHR
.186	DW	GE	MASK	SIZE
.286	DWORD	GROUP	MOD	.STACK
.286P	ELSE	GT	.MODEL	STRUC
.287	END	HIGH	NAME	SUBTTL
.386	ENDIF	IF	NE	TBYTE
.386P	ENDM	IF1	NEAR	.TFCOND
.387	ENDP	IF2	NOT	THIS
.8086	ENDS	IFB	OFFSET	TITLE
.8087	EQ	IFDEF	OR	TYPE
ALIGN	EQU	IFDIF	ORG	.TYPE
.ALPHA	.ERR	IFE	%OUT	WIDTH
AND	.ERR1	IFIDN	PAGE	WORD
ASSUME	.ERR2	IFNB	PROC	.XALL
BYTE	.ERRB	IFNDEF	PTR	.XCREF
.CODE	.ERRDEF	INCLUDE	PUBLIC	.XLIST
COMM	.ERRDIF	INCLUDELIB	PURGE	XOR
COMMENT	.ERRE	IRP	QWORD	
.CONST	.ERRIDN	IRPC	.RADIX	
.CREF	.ERRNB	LABEL	RECORD	

In addition to these names in the table above, instruction mnemonics and register names are considered reserved names. These vary depending on the processor directives given in the source file. For example, the register name **EAX** is a reserved word with the **.386** directive but not with the **.286** directive. Section 4.4.1, "Defining Default Assembly Behavior," describes processor directives. Instruction mnemonics for each processor are listed in the *Microsoft Macro Assembler Reference*. Register names are listed in Section 14.2, "Using Register Operands."

4.3 Constants

Constants can be used in source files to specify numbers or strings that are set or initialized at assembly time. **MASM** recognizes four types of constant values:

1. Integers
2. Packed binary coded decimals
3. Real numbers
4. Strings

4.3.1 Integer Constants

Integer constants represent integer values. They can be used in a variety of contexts in assembly-language source code. For example, they can be used in data declarations and equates, or as immediate operands.

Packed decimal integers are a special kind of integer constant that can only be used to initialize binary coded decimal (BCD) variables. They are described in Sections 4.3.2, "Packed Binary Coded Decimal Constants," and 6.2.1.2, "Binary Coded Decimal Variables."

Integer constants can be specified in binary, octal, decimal, or hexadecimal values. Table 4.2 shows the legal digits for each of these radixes. For hexadecimal radix, the digits can be either uppercase or lowercase letters.

Table 4.2

Digits Used with Each Radix

Name	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

The radix for an integer can be defined for a specific integer by using radix specifiers; or a default radix can be defined globally with the **.RADIX** directive.

4.3.1.1 Specifying Integers with Radix Specifiers

The radix for an integer constant can be given by putting one of the following radix specifiers after the last digit of the number:

Radix	Specifier
Binary	**B**
Octal	**Q** or **O**
Decimal	**D**
Hexadecimal	**H**

Radix specifiers can be given in either uppercase or lowercase letters; sample code in this manual uses lowercase letters.

Hexadecimal numbers must always start with a decimal digit (0 to 9). If necessary, put a leading 0 at the left of the number to distinguish between symbols and hexadecimal numbers that start with a letter. For example, 0ABCh is interpreted as a hexadecimal number, but ABCh is interpreted as a symbol. The hexadecimal digits A through F can be either uppercase or lowercase letters. Sample code in this manual uses uppercase letters.

If no radix is given, the assembler interprets the integer by using the current default radix. The initial default radix is decimal, but you can change the default with the **.RADIX** directive.

■ Examples

```
n360        EQU    01011010b + 132q + 5Ah + 90d ; 4 * 90
n60         EQU    00001111b +  17o + 0Fh + 15d ; 4 * 15
```

4.3.1.2 Setting the Default Radix

The **.RADIX** directive sets the default radix for integer constants in the source file.

■ Syntax

.RADIX *expression*

The *expression* must evaluate to a number in the range 2–16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base.

Numbers given in *expression* are always considered decimal, regardless of the current default radix. The initial default radix is decimal.

Note

The **.RADIX** directive does not affect real numbers initialized as variables with the **DD**, **DQ**, or **DT** directive. Initial values for variables declared with these directives are always evaluated as decimal unless a radix specifier is appended.

Also, the **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integer numbers. When the letters **B** or **D** appear at the end of any integer, they are always considered to be a radix specifier even if the current radix is 16.

For example, if the input radix is 16, the number `0ABCD` will be interpreted as 0ABC decimal, an illegal number, instead of as 0ABCD hexadecimal, as intended. Type `0ABCDh` to specify 0ABCD in hexadecimal. Similarly, the number `11B` will be treated as 11 binary, a legal number, but not as 11B hexadecimal as intended. Type `11Bh` to specify 11B in hexadecimal.

■ **Examples**

```
.RADIX  16          ; Set default radix to hexadecimal
.RADIX  2           ; Set default radix to binary
```

4.3.2 Packed Binary Coded Decimal Constants

When an integer constant is used with the **DT** directive, the number is interpreted by default as a packed binary coded decimal number. You can use the **D** radix specifier to override the default and initialize 10-byte integers as binary-format integers.

The syntax for specifying binary coded decimals is exactly the same as for other integers. However, **MASM** encodes binary coded decimals in a completely different way. See Section 6.2.1.2, "Defining Binary Coded Decimal Variables," for complete information on storage of binary coded decimals.

■ **Examples**

```
positive    DT      1234567890  ; Encoded as 0000000000001234567890h
negative    DT      -1234567890 ; Encoded as 8000000000001234567890h
```

4.3.3 Real-Number Constants

A real number is a number consisting of an integer part, a fractional part, and an exponent. Real numbers are usually represented in decimal format.

■ Syntax

[[+ | −]] *integer.fraction*[[**E**[[+ | −]]*exponent*]]

The *integer* and *fraction* parts combine to form the value of the number. This value is stored internally as a unit and is called the mantissa. It may be signed. The optional *exponent* follows the exponent indicator (**E**). It represents the magnitude of the value, and is stored internally as a unit. If no *exponent* is given, 1 is assumed. If an exponent is given, it may be signed.

During assembly, **MASM** converts real-number constants given in the decimal format to a binary format. The sign, exponent, and mantissa of the real number are encoded as bit fields within the number. See Section 6.3.1.5, "Real-Number Variables," for an explanation of how real numbers are encoded.

You can specify the encoded format directly using hexadecimal digits (0–9 or A–F). The number must begin with a decimal digit (0–9) and cannot be signed. It must be followed by the real-number designator (**R**). This designator is used the same as a radix designator except it specifies that the given hexadecimal number should be interpreted as a real number.

Real numbers can only be used to initialize variables with the **DD**, **DQ**, and **DT** directives. They cannot be used in expressions. The maximum number of digits in the number and the maximum range of exponent values depend on the directive. The number of digits for encoded numbers used with **DD**, **DQ**, and **DT** must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See Section 6.3.1.5, "Real-Number Variables," for an explanation of how real numbers are encoded.

Note

Real numbers will be encoded differently depending upon whether you use the **.MSFLOAT** directive. By default, real numbers are encoded in the IEEE format. This is a change from previous versions, which assembled real numbers by default in the Microsoft Binary format. The **.MSFLOAT** directive overrides the default and specifies Microsoft Binary format. See Section 6.3.1.5, "Real-Number Variables," for a description of these formats.

■ Example

```
            ; Real numbers
shrt        DD      25.23
long        DQ      2.523E1
ten_byte    DT      2523.0E-2

            ; Assumes .MSFLOAT
mbshort     DD      81000000r               ; 1.0 as Microsoft Binary short
mblong      DQ      8100000000000000r       ; 1.0 as Microsoft Binary long

            ; Assumes default IEEE format
ieeeshort   DD      3F800000r               ; 1.0 as IEEE short
ieeelong    DQ      3FF0000000000000r       ; 1.0 as IEEE long

            ; The same regardless of processor directives
temporary   DT      3FFF8000000000000000r ; 1.0 as 10-byte temporary real
```

4.3.4 String Constants

A string constant consists of one or more ASCII characters enclosed in single or double quotation marks.

■ Syntax

'characters'
"characters"

String constants are case sensitive. A string constant consisting of a single character is sometimes called a character constant.

Single quotation marks must be encoded twice when used literally within string constants that are also enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when used in string constants that are also enclosed by double quotation marks.

■ Examples

```
char        DB      'a'
char2       DB      "a"
message     DB      "This is a message."
warn        DB      'Can''t find file.'         ; Can't find file.
warn2       DB      "Can't find file."          ; Can't find file.
string      DB      "This ""value"" not found." ; This "value" not found.
string2     DB      'This "value" not found.'   ; This "value" not found.
```

4.4 Defining Default Assembly Behavior

Since the assembler processes sequentially, any directives that define the behavior of the assembler for sections of code or for the entire source file must come before the sections affected by the directive.

There are three types of directives that may define behavior for the assembly:

1. The **.MODEL** directive defines the memory model.
2. Processor directives define the processor and coprocessor.
3. The **.MSFLOAT** directive and the coprocessor directives define how floating-point variables are encoded.

These directives are optional. If you do not use them, **MASM** makes default assumptions. However, if you do use them, you must put them before any statements that will be affected by them.

The **.MSFLOAT** and **.MODEL** directives affect the entire assembly and can only occur once in the source file. Normally they should be placed at the beginning of the source file.

The **.MODEL** directive is part of the new system of simplified segment directives implemented in Version 5.0. It is explained in Section 5.1.3., "Defining the Memory Model."

The **.MSFLOAT** directive disables all coprocessor instructions and specifies that initialized real-number variables be encoded in the Microsoft Binary format. Without this directive, initialized real-number variables are encoded in the IEEE format. This is a change from previous versions of the assembler, which used Microsoft Binary format by default and required a coprocessor directive or the **/R** option to specify IEEE format. **.MSFLOAT** must be used for programs that require real-number data in the Microsoft Binary format. Section 6.3.1.5, "Real-Number Variables," describes real-number data formats and the factors to consider in choosing a format.

Processor and coprocessor directives define the instruction set that is recognized by **MASM**. They are listed and explained below:

Directive	Description
.8086	The **.8086** directive enables assembly of instructions for the 8086 and 8088 processors and the 8087 coprocessor. It disables assembly of the instructions unique to the 80186, 80286, and 80386 processors.

This is the default mode and is used if no instruction set directive is specified. Using the default instruction set ensures that your program can be used on all 8086-family processors. However, if you choose this directive, your program will not take advantage of the more powerful instructions available on more advanced processors.

.186 The **.186** directive enables assembly of the 8086 processor instructions, 8087 coprocessor instructions, and the additional instructions for the 80186 processor.

.286 The **.286** directive enables assembly of the 8086 instructions plus the additional nonprivileged instructions of the 80286 processor. It also enables 80287 coprocessor instructions. If privileged instructions were previously enabled, the **.286** directive disables them.

This directive should be used for programs that will be executed only by an 80186, 80286, or 80386 processor. For compatibility with previous versions of **MASM**, the **.286C** directive is also available. It is equivalent to the **.286** directive.

.286P This directive is equivalent to the **.286** directive except that it also enables the privileged instructions of the 80286 processor. This does not mean that the directive is required if the program will run in protected mode; it only means that the directive is required if the program uses the instructions that initiate and manage privileged-mode processes. These instructions (see Section 20.3, "Controlling Protected Mode Processes") are normally used only by systems programmers.

.386 The **.386** directive enables assembly of the 8086 and the nonprivileged instructions of the 80286 and 80386 processors. It also enables 80387 coprocessor instructions. If privileged instructions were previously enabled, this directive disables them.

This directive should be used for programs that will be executed only by an 80386 processor.

.386P This directive is equivalent to the **.386** directive except that it also enables the privileged instructions of the 80386 processor.

.8087 The **.8087** directive enables assembly of instructions for the 8087 math coprocessor and disables assembly of instructions unique to the 80287 coprocessor. It also specifies the IEEE format for encoding floating-point variables.

This is the default mode and is used if no coprocessor directive is specified. This directive should be used for programs that must run with either the 8087, 80287, or 80387 coprocessors.

.287 The **.287** directive enables assembly of instructions for the 8087 floating-point coprocessor and the additional instructions for the 80287. It also specifies the IEEE format for encoding floating-point variables.

Coprocessor instructions are optimized if you use this directive rather than the **.8087** directive. Therefore, you should use it if you know your program will never need to run under an 8087 processor. See Section 19.3, "Coordinating Memory Access," for an explanation.

.387 The **.387** directive enables assembly of instructions for the 8087 and 80287 floating-point coprocessors and the additional instructions and addressing modes for the 80387. It also specifies the IEEE format for encoding floating-point variables.

If you do not specify any processor directives, **MASM** uses the following defaults:

- 8086/8088 processor instruction set
- 8087 coprocessor instruction set
- IEEE format for floating-point variables

Normally the processor and coprocessor directives can be used at the start of the source file to define the instruction sets for the entire assembly. However, it is possible to use different processor directives at different points in the source file to change assumptions for a section of code. For instance, you might have processor-specific code in different parts of the same source file. You can also turn privileged instructions on and off or allow unusual combinations of the processor and coprocessor.

There are two limitations on changing the processor or coprocessor:

1. The directives must be given outside segments. You must end the current segment, give the processor directive, and then open

another segment. See Section 5.1.5, "Using Predefined Equates," for an example of changing the processor directives with simplified segment directives.

2. You can specify a lower-level coprocessor with a higher-level coprocessor, but an error message will be generated if you try to specify a lower-level processor with a higher-level coprocessor.

The coprocessor directives have the opposite effect of the **.MSFLOAT** directive. **.MSFLOAT** turns off coprocessor instruction sets and enables the Microsoft Binary format for floating-point variables. Any coprocessor instruction turns on the specified coprocessor instruction set and enables IEEE format for floating-point variables.

■ Examples

```
; .MSFLOAT affects the whole source file
            .MSFLOAT
            .8087                ; Ignored

; Legal - use 80386 and 80287
            .386
            .287

; Illegal - can't use 8086 with 80287
            .8086
            .287

; Turn privileged mode on and off
            .286P
            .
            .
            .
            .286
```

4.5 Ending a Source File

Source files are always terminated with the **END** directive. This directive has two purposes: it marks the end of the source file, and it can indicate the address where execution begins when the program is loaded.

■ Syntax

END [*startaddress*]

Any statements following the **END** directive are ignored by the assembler. For instance, you can put comments after the **END** directive without using comment specifiers (;) or the **COMMENT** directive.

The *startaddress* is a label or expression identifying the address where you want execution to begin when the program is loaded. Specifying a start address is discussed in detail in Section 5.5.1, "Initializing the CS and IP Registers."

CHAPTER 5

DEFINING SEGMENT STRUCTURE

5.1 Simplified Segment Definitions 83
- 5.1.1 Understanding Memory Models 84
- 5.1.2 Specifying DOS Segment Order 85
- 5.1.3 Defining the Memory Model 87
- 5.1.4 Defining Simplified Segments 88
- 5.1.5 Using Predefined Equates 90
- 5.1.6 Simplified Segment Defaults 92
- 5.1.7 Default Segment Names 93

5.2 Full Segment Definitions 95
- 5.2.1 Setting the Segment-Order Method 96
- 5.2.2 Defining Full Segments 97
  - 5.2.2.1 Controlling Alignment with Align Type 98
  - 5.2.2.2 Setting Segment Word Size with Use Type 98
  - 5.2.2.3 Defining Segment Combinations with Combine Type 100
  - 5.2.2.4 Controlling Segment Structure with Class Type 104

5.3 Defining Segment Groups 106

5.4 Associating Segments with Registers 109

5.5 Initializing Segment Registers 111
- 5.5.1 Initializing the CS and IP Registers 111
- 5.5.2 Initializing the DS Register 112
- 5.5.3 Initializing the SS and SP Registers 114
- 5.5.4 Initializing the ES Register 115

5.6 Nesting Segments 115

Segments are a fundamental part of assembly-language programming for the 8086-family of processors. They are related to the segmented architecture used by Intel® for its 16-bit and 32-bit microprocessors. This architecture is explained in more detail in Chapter 13, "Understanding 8086-Family Processors."

A segment is a collection of instructions or data whose addresses are all relative to the same segment register. Segments can be defined by using simplified segment directives or full segment definitions.

In most cases, simplified segment definitions are a better choice. They are easier to use and more consistent, yet you seldom sacrifice any functionality by using them. Simplified segment directives automatically define the segment structure required when combining assembler modules with modules prepared with Microsoft high-level languages.

Although more difficult to use, full segment definitions give more complete control over segments. A few complex programs may require full segment definitions in order to get unusual segment orders and types. In previous versions of **MASM**, full segment definitions are the only way to define segments, so you may need to use them to maintain existing source code.

This chapter describes both methods. If you choose to use simplified segment directives, you will probably not need to read about full segment definitions.

5.1 Simplified Segment Definitions

Version 5.0 of **MASM** implements a new simplified system for declaring segments. By default, the simplified segment directives use the segment names and conventions followed by Microsoft high-level languages. If you are willing to accept these conventions, the more difficult aspects of segment definition are handled automatically.

If you are writing stand-alone assembler programs in which segment names, order, and other definition factors are not crucial, the simplified segment directives make programming easier. The Microsoft conventions are flexible enough to work for most kinds of programs. If you are new to assembly-language programming, you should use the simplified segment directives for your first programs.

If you are writing assembler routines to be linked with Microsoft high-level languages, the simplified segment directives ensure against mistakes that would make your modules incompatible. The names are automatically defined consistently and correctly.

When you use simplified segment directives, **ASSUME** and **GROUP** statements that are consistent with Microsoft conventions are generated

automatically. You can learn more about the **ASSUME** and **GROUP** directives in Sections 5.3 and 5.4. However, for most programs you do not need to understand these directives. You simply use the simplified segment directives in the format shown in the examples.

Note

The simplified segment directives cannot be used for programs written in the **.COM** format. You must specifically define the single segment required for this format. See Section 1.4.1, "Writing and Editing Assembly-Language Source Code," for more information.

5.1.1 Understanding Memory Models

To use simplified segment directives, you must declare a memory model for your program. The memory model specifies the default size of data and code used in a program.

Microsoft high-level languages require that each program have a default size (or memory model). Any assembly-language routine called from a high-level-language program should have the same memory model as the calling program. See the documentation for your language to find out what memory models it can use.

The most commonly used memory models are described below:

Model	Description
Tiny	All data and code fits in a single segment. Tiny model programs must be written in the **.COM** format. Microsoft languages do not support this model. Some compilers from other companies support tiny model either as an option or as a requirement. You cannot use simplified segment directives for tiny-model programs.
Small	All data fits within a single 64K segment, and all code fits within a 64K segment. Therefore, all code and data can be accessed as near. This is the most common model for stand-alone assembler programs. C is the only Microsoft language that supports this model.
Medium	All data fits within a single 64K segment, but code may be greater than 64K. Therefore, data is near, but code is far. Most recent versions of Microsoft languages support this model.
Compact	All code fits within a single 64K segment, but the total amount of data may be greater than 64K (although no

array can be larger than 64K). Therefore, code is near, but data is far. C is the only Microsoft language that supports this model.

Large — Both code and data may be greater than 64K (although no array can be larger than 64K). Therefore, both code and data are far. All Microsoft languages support this model.

Huge — Both code and data may be greater than 64K. In addition, data arrays may be larger than 64k. Both code and data are far, and pointers to elements within an array must also be far. Most recent versions of Microsoft languages support this model. Segments are the same for large and huge models.

Stand-alone assembler programs can have any model. Small model is adequate for most programs written entirely in assembly language. Since near data or code can be accessed more quickly, the smallest memory model that can accommodate your code and data is usually the most efficient.

Mixed-model programs use the default size for most code and data but override the default for particular data items. Stand-alone assembler programs can be written as mixed-model programs by making specific procedures or variables near or far. Some Microsoft high-level languages have **NEAR**, **FAR**, and **HUGE** keywords that enable you to override the default size of individual data or code items.

5.1.2 Specifying DOS Segment Order

The **DOSSEG** directive specifies that segments be ordered according to the DOS segment-order convention. This is the convention used by Microsoft high-level-language compilers.

■ Syntax

DOSSEG

Using the **DOSSEG** directive enables you to maintain a consistent, logical segment order without actually defining segments in that order in your source file. Without this directive, the final segment order of the executable file depends on a variety of factors, such as segment order, class name, and order of linking. These factors are described in Section 5.2, "Full Segment Definitions."

Since segment order is not crucial to the proper functioning of most stand-alone assembler programs, you can simply use the **DOSSEG** directive and ignore the whole issue of segment order.

Note

Using the **DOSSEG** directive (or the **/DOSSEG** linker option) has two side effects. The linker generates symbols called **_end** and **_edata**. You should not use these names in programs that contain the **DOSSEG** directive. Also, the linker increases the offset of the first byte of the code segment by 16 bytes in small and compact models. This is to give proper alignment to executable files created with Microsoft compilers.

If you want to use the DOS segment-order convention in stand-alone assembler programs, you should use the **DOSSEG** argument in the main module. Modules called from the main module need not use the **DOSSEG** directive.

You do not need to use the **DOSSEG** directive for modules called from Microsoft high-level languages, since the compiler already defines DOS segment order.

Under the DOS segment-order convention, segments have the following order:

1. All segment names having the class name **'CODE'**
2. Any segments that do not have class name **'CODE'** and are not part of the group **DGROUP**
3. Segments that are part of **DGROUP**, in the following order:
   a. Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)
   b. Any segments not of class **BEGDATA**, **BSS**, or **STACK**
   c. Segments of class **BSS**
   d. Segments of class **STACK**

Using the **DOSSEG** directive has the same effect as using the **/DOSSEG** linker option.

The directive works by writing to the comment record of the object file. The Intel title for this record is **COMENT**. If the linker detects a certain sequence of bytes in this record, it automatically puts segments in the DOS order.

5.1.3 Defining the Memory Model

The **.MODEL** directive is used to initialize the memory model. This directive should be used early in the source code before any other segment directive.

■ Syntax

.MODEL *memorymodel*

The *memorymodel* can be **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, or **HUGE**. Segments are defined the same for large and huge models, but the **@datasize** equate (explained in Section 5.1.5, "Using Predefined Equates") is different.

If you are writing an assembler routine for a high-level language, the *memorymodel* should match the memory model used by the compiler or interpreter.

If you are writing a stand-alone assembler program, you can use any model. Section 5.1.1 describes each memory model. Small model is the best choice for most stand-alone assembler programs.

Note

You must use the **.MODEL** directive before defining any segment. If one of the other simplified segment directives (such as **.CODE** or **.DATA**) is given before the **.MODEL** directive, an error is generated.

■ Example 1

```
DOSSEG
.MODEL  small
```

This statement defines default segments for small-model programs and creates the **ASSUME** and **GROUP** statements used by small-model programs. The segments are automatically ordered according to the Microsoft convention. The example statements might be used at the start of the main (or only) module of a stand-alone assembler program.

■ Example 2

```
.MODEL  LARGE
```

This statement defines default segments for large-model programs and creates the **ASSUME** and **GROUP** statements used by large-model programs. It does not automatically order segments according to the Microsoft convention. The example statement might be used at the start of an assembly module that would be called from a large-model C, BASIC, FORTRAN, or Pascal program.

■ 80386 Only

If you use the **.386** directive before the **.MODEL** directive, the segment definitions defines 32-bit segments. If you want to enable the 80386 processor with 16-bit segments, you should give the **.386** directive after the **.MODEL** directive.

5.1.4 Defining Simplified Segments

The **.CODE**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.CONST**, and **.STACK** directives indicate the start of a segment. They also end any open segment definition used earlier in the source code.

■ Syntax

.STACK [[*size*]]	Stack segment
.CODE [[*name*]]	Code segment
.DATA	Initialized near-data segment
.DATA?	Uninitialized near-data segment
.FARDATA [[*name*]]	Initialized far-data segment
.FARDATA? [[*name*]]	Uninitialized far-data segment
.CONST	Constant-data segment

For segments that take an optional *name*, a default name is used if none is specified. See Section 5.1.7 for information on default segment names.

Each new segment directive ends the previous segment. The **END** directive closes the last open segment in the source file.

The *size* argument of the **.STACK** directive is the number of bytes to be declared in the stack. If no *size* is given, the segment is defined with a default size of one kilobyte.

Stand-alone assembler programs in the **.EXE** format should define a stack for the main (or only) module. Stacks are defined by the compiler or interpreter for modules linked with a main module from a high-level language.

Code should be placed in a segment initialized with the **.CODE** directive, regardless of the memory model. Normally, only one code segment is defined in a source module. If you put multiple code segments in one source file, you must specify *name* to distinguish the segments. The *name* can only be specified for models allowing multiple code segments (medium and large). *Name* will be ignored if given with small or compact models.

Uninitialized data is any variable declared by using the indeterminate symbol (**?**) and the **DUP** operator. When declaring data for modules that will be used with a Microsoft high-level language, you should follow the convention of using **.DATA** or **.FARDATA** for initialized data and **.DATA?** or **.FARDATA?** for uninitialized data. For stand-alone assembler programs, using the **.DATA?** and **.FARDATA?** directives is optional. You can put uninitialized data in any data segment.

Constant data is data that must be declared in a data segment but is not subject to change at run time. Use of this segment is optional for stand-alone assembler programs. If you are writing assembler routines to be called from a high-level language, you can use the **.CONST** directive to declare strings, real numbers, and other constant data that must be allocated as data.

Data in segments defined with the **.STACK**, **.CONST**, **.DATA** or **.DATA?** directives is placed in a group called **DGROUP**. Data in segments defined with the **.FARDATA** or **.FARDATA?** directives is not placed in any group. See Section 5.3 for more information on segment groups. When initializing the **DS** register to access data in a group-associated segment, the value of **DGROUP** should be loaded into **DS**. See Section 5.5.2 for information on initializing data segments.

■ Example 1

```
                DOSSEG
                .MODEL  SMALL
                .STACK  100h
                .DATA
ivariable       DB      5
iarray          DW      50 DUP (5)
string          DB      "This is a string"
uarray          DW      50 DUP (?)
                EXTRN   xvariable:WORD
                .CODE
start:          mov     ax,DGROUP
                mov     ds,ax
                EXTRN   xprocedure:NEAR
                call    xprocedure
                .
                .
                .
                END     start
```

This code uses simplified segment directives for a small-model, stand-alone assembler program. Notice that initialized data, uninitialized data, and a string constant are all defined in the same data segment. See Section 5.1.7, "Default Segment Names," for an equivalent version that uses full segment definitions.

■ Example 2

```
            .MODEL  LARGE
            .FARDATA?
fuarray     DW      10 DUP (?)          ; Far uninitialized data
            .CONST
string      DB      "This is a string" ; String constant
            .DATA
niarray     DB      100 DUP (5)         ; Near initialized data
            .FARDATA
            EXTR    xvariable:FAR
fiarray     DW      100 DUP (10)        ; Far initialized data
            .CODE   ACTION
            EXTR    xprocedure:PROC
task        PROC
            .
            .
            .
            ret
task        ENDP
            END
```

This example uses simplified segment directives to create a module that might be called from a large-model, high-level-language program. Notice that different types of data are put in different segments to conform to Microsoft compiler conventions. See Section 5.1.7, "Default Segment Names," for an equivalent version using full segment definitions.

5.1.5 Using Predefined Equates

Several equates are predefined for you. You can use the equate names at any point in your code to represent the equate values. You should not assign equates having these names. The predefined equates are listed below:

Name	Value
@curseg	This name has the segment name of the current segment. This value may be convenient for **ASSUME** statements, segment overrides, or other cases in which you need to access the current segment. It can also be used to end a segment, as shown below:

```
@curseg ENDS      ; End current segment
        .286      ; Must be outside segment
        .CODE     ; Restart segment
```

`@filename`

This value represents the base name of the current source file. For example, if the current source file is `task.asm`, the value of `@filename` is `task`. This value can be used in any name you would like to change if the file name changes. For example, it can be used as a procedure name:

```
@filename  PROC
           .
           .
           .
@filename  ENDP
```

`@codesize` and `@datasize`

If the **.MODEL** directive has been used, the `@codesize` value is 0 for small and compact models or 1 for medium, large, and huge models. The `@datasize` value is 0 for small and medium models, 1 for compact and large models, and 2 for huge models. These values can be used in conditional-assembly statements:

```
IF      @datasize
les     bx,pointer          ; Load far pointer
mov     ax,es:WORD PTR [bx]
ELSE
mov     bx,WORD PTR pointer ; Load near pointer
mov     ax,WORD PTR [bx]
ENDIF
```

Segment equates

For each of the primary segment directives, there is a corresponding equate with the same name, except that the equate starts with an at sign (@) but the directive starts with a period. For example, the `@code` equate represents the segment name defined by the **.CODE** directive. Similarly, `@fardata` represents the **.FARDATA** segment name and `@fardata?` represents the **.FARDATA?** segment name. The `@data` equate represents the group name shared by all the near data segments. It can be used to access the segments created by the **.DATA**, **.DATA?**, **.CONST**, and **.STACK** segments.

These equates can be used in **ASSUME** statements and at any other time a segment must be referred to by name, for example:

```
ASSUME es:@fardata ; Assume ES to far data
                   ;   (.MODEL handles DS)
mov     ax,@data   ; Initialize near to DS
mov     ds,ax
mov     ax,@fardata ; Initialize far to ES
mov     es,ax
```

Note

Although predefined equates are part of the simplified segment system, the `@curseg` and `@filename` equates are also available when using full segment definitions.

5.1.6 Simplified Segment Defaults

When you use the simplified segment directives, defaults are different in certain situations than they would be if you gave full segment definitions. Defaults that change are listed below:

- If you give full segment definitions, the default size for the **PROC** directive is always **NEAR**. If you use the **.MODEL** directive, the **PROC** directive is associated with the specified memory model: **NEAR** for small and compact models and **FAR** for medium, large, and huge models. See Section 6.1.2, "Procedure Labels," for further discussion of the **PROC** directive.

- If you give full segment definitions, the segment address used as the base when calculating an offset with the **OFFSET** operator is the data segment (the segment associated with the **DS** register). With the simplified segment directives, the base address is the **DGROUP** segment for segments that are associated with a group. This includes segments declared with the **.DATA**, **.DATA?**, and **.STACK** directives, but not segments declared with the **.CODE**, **.FARDATA**, and **.FARDATA?** directives.

  For example, assume the variable `test1` was declared in a segment defined with the **.DATA** directive and `test2` was declared in a segment defined with the **.FARDATA** directive. The statement

  ```
  mov     ax,OFFSET test1
  ```

  loads the address of `test1` relative to **DGROUP**. The statement

  ```
  mov     ax,OFFSET test2
  ```

  loads the address of `test2` relative to the segment defined by the **.FARDATA** directive. See Section 5.3 for more information on groups.

5.1.7 Default Segment Names

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, it is possible to mix full segment definitions with simplified segment definitions. Therefore, some programmers may wish to know the actual names assigned to all segments.

Table 5.1 shows the default segment names created by each directive.

Table 5.1

Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Small	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Medium	.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Compact	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Large	.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
or huge	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP

The *name* used as part of far-code segment names is the file name of the module. The default name associated with the **.CODE** directive can be overridden in medium and large models. The default names for the **.FARDATA** and **.FARDATA?** directives can always be overridden.

The segment and group table at the end of listings always shows the actual segment names. However, the group and assume statements generated by the **.MODEL** directive are not shown in listing files. For a program that uses all possible segments, group statements equivalent to the following would be generated:

```
DGROUP  GROUP   _DATA,CONST,_BSS,STACK
```

For small and compact models, the following would be generated:

```
        ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
```

For medium, large, and huge models the following statement is given:

```
        ASSUME  cs:name_TEXT,ds:DGROUP,ss:DGROUP
```

80386 Only

If the **.386** directive is used, the default align type for all segments is **DWORD**.

■ Example 1

```
            EXTRN   xvariable:WORD
            EXTRN   xprocedure:NEAR
DGROUP      GROUP   _DATA,_BSS
            ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT       SEGMENT WORD PUBLIC 'CODE'
start:      mov     ax,DGROUP
            mov     ds,ax
            .
            .
            .
_TEXT       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
ivariable   DB      5
iarray      DW      50 DUP (5)
string      DB      "This is a string"
uarray      DW      50 DUP (?)
_DATA       ENDS
STACK       SEGMENT PARA STACK 'STACK'
            DB      100h DUP (?)
STACK       ENDS
            END     start
```

This example is equivalent to Example 1 in Section 5.1.4, "Defining Simplified Segments." Notice that the segment order must be different in this version to achieve the segment order specified by using the **DOSSEG** directive in the first example. The external variables are declared at the start of the source code in this example. With simplified segment directives, they can be declared in the segment in which they are used.

■ Example 2

```
DGROUP      GROUP   _DATA,CONST,STACK
            ASSUME  cs:TASK_TEXT,ds:FAR_DATA,ss:STACK
            EXTRN   xprocedure:FAR
            EXTR    xvariable:FAR
FAR_BSS     SEGMENT PARA 'FAR_DATA'
fuarray     DW      10 DUP (?)          ; Far uninitialized data
FAR_BSS     ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
string      DB      "This is a string" ; String constant
CONST       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
niarray     DB      100 DUP (5)         ; Near initialized data
_DATA       ENDS
FAR_DATA    SEGMENT WORD 'FAR_DATA'
fiarray     DW      100 DUP (10)
FAR_DATA    ENDS
TASK_TEXT   SEGMENT WORD PUBLIC 'CODE
task        PROC    FAR
            .
            .
            .
            ret
task        ENDP
TASK_TEXT   ENDS
            END
```

This example is equivalent to Example 2 in Section 5.1.4, "Defining Simplified Segments." Notice that the segment order is the same in both versions. The segment order shown here is written to the object file, but it is different in the executable file. The segment order specified by the compiler (the DOS segment order) overrides the segment order in the module object file.

5.2 Full Segment Definitions

If you need complete control over segments, you may want to give complete segment definitions. The section below explains all aspects of segment definitions, including how to order segments and how to define all the segment types.

5.2.1 Setting the Segment-Order Method

The order in which **MASM** writes segments to the object file can be either sequential or alphabetical. If the sequential method is specified, segments are written in the order in which they appear in the source code. If the alphabetical method is specified, segments are written in the alphabetical order of their segment names.

The default is sequential. If no segment-order directive or option is given, segments are ordered sequentially. The segment-order method is only one factor in determining the final order of segments in memory. The **DOSSEG** directive (see Section 5.1.2, "Specifying DOS Segment Order") and class type (see Section 5.2.2.4, "Controlling Segment Structure with Class Type") can also affect segment order.

The ordering method can be set by using the **.ALPHA** or **.SEQ** directive in the source code. The method can also be set using the **/S** (sequential) or **/A** (alphabetical) assembler options (see Section 2.4.1, "Specifying the Segment-Order Method"). The directives have precedence over the options. For example, if the source code contains the **.ALPHA** directive, but the **/S** option is given on the command line, the segments are ordered alphabetically.

Changing the segment order is an advanced technique. In most cases you can simply leave the default sequential order in effect. If you are linking with high-level-language modules, the compiler automatically sets the segment order. The **DOSSEG** directive also overrides any segment-order directives or options.

Note

Some previous versions of the IBM Macro Assembler ordered segments alphabetically by default. If you have trouble assembling and linking source-code listings from books or magazines, try using the **/A** option. Listings written for previous IBM versions of the assembler may not work without this option.

■ **Example 1**

```
            .SEQ
DATA        SEGMENT WORD PUBLIC 'DATA'
DATA        ENDS
CODE        SEGMENT WORD PUBLIC 'CODE'
CODE        ENDS
```

■ **Example 2**

```
            .ALPHA
DATA        SEGMENT WORD PUBLIC 'DATA'
DATA        ENDS
CODE        SEGMENT WORD PUBLIC 'CODE'
CODE        ENDS
```

In Example 1, the DATA segment is written to the object file first because it appears first in the source code. In Example 2, the CODE segment is written to the object file first because its name comes first alphabetically.

5.2.2 Defining Full Segments

The beginning of a program segment is defined with the **SEGMENT** directive, and the end of the segment is defined with the **ENDS** directive.

■ **Syntax**

name **SEGMENT** [[*align*]] [[*combine*]] [[*use*]] [['*class*']]
statements
name **ENDS**

The *name* defines the name of the segment. This name can be unique or it can be the same name given to other segments in the program. Segments with identical names are treated as the same segment. For example, if it is convenient to put different portions of a single segment in different source modules, the segment is given the same name in both modules.

The optional *align*, *combine*, *use*, and '*class*'

types give the linker and the assembler instructions on how to set up and combine segments. Types should be specified in order, but it is not necessary to enter all types, or any type, for a given segment.

Defining segment types is an advanced technique. Beginning assembly-language programmers might try using the simplified segment directives discussed in Section 5.1.

Note

Don't confuse the **PAGE** align type and the **PUBLIC** combine type with the **PAGE** and **PUBLIC** directives. The distinction should be clear from context since the align and combine types are only used on the same line as the **SEGMENT** directive.

Segment types have no effect on programs prepared in the **.COM** format. Since there is only one segment, there is no need to specify how segments are combined or ordered.

5.2.2.1 Controlling Alignment with Align Type

The optional *align* type defines the range of memory addresses from which a starting address for the segment can be selected. The *align* type can be any one of the following:

Align Type	Meaning
BYTE	Uses the next available byte address.
WORD	Uses the next available word address (2 bytes per word).
DWORD	Uses the next available doubleword address (4 bytes per doubleword); the **DWORD** align type is normally used in 32-bit segments with the 80386.
PARA	Uses the next available paragraph address (16 bytes per paragraph).
PAGE	Uses the next available page address (256 bytes per page).

If no *align* type is given, **PARA** is used by default (except with the 80386).

The linker uses the alignment information to determine the relative start address for each segment. DOS uses the information to calculate the actual start address when the program is loaded.

Align types are illustrated in Figure 5.1, in Section 5.2.2.3, "Defining Segment Combinations with Combine Type."

5.2.2.2 Setting Segment Word Size with Use Type

■ **80386 Only**

The *use* type specifies the segment word size on the 80386 processor. Segment word size is the default operand and address size of a segment.

The *use* type can be **USE16** or **USE32**. These types are only relevant if you have enabled 80386 instructions and addressing modes with the **.386** directive. The assembler generates an error if you specify use *type* when the 80386 processor is not enabled.

With the 80286 and other 16-bit processors, the segment word size is always 16 bits. A 16-bit segment can contain up to 65,536 (64K) bytes. However, the 80386 is capable of using either 16-bit or 32-bit segments. A 32-bit segment can contain up to 4,294,967,296 bytes (4 gigabytes). Although **MASM** permits you to define 4 gigabyte segments in 32-bit segments, current versions of DOS limit segment size to 64K.

If you do not specify a *use* type, the segment word size is 32 bits by default when the **.386** directive is used.

The effect of addressing modes is changed by the word size you specify for the code segment. See Section 14.3.3, "80386 Indirect Memory Operands," for more information on 80386 addressing modes. The meaning of the **WORD** and **DWORD** type specifiers is not changed by the *use* type. **WORD** always indicates 16 bits and **DWORD** always indicates 32 bits regardless of the current segment word size.

Note

Although the assembler allows you to use 16-bit and 32-bit segments in the same program, you should normally make all segments the same size. Mixing segment sizes is an advanced technique that can have unexpected side effects. For the most part, it is used only by systems programmers.

■ Example 1

```
; 16-bit segment
            .386
_DATA       SEGMENT DWORD USE16 PUBLIC 'DATA'
            .
            .
            .
_DATA       ENDS
```

■ Example 2

```
; 32-bit segment
_TEXT       SEGMENT DWORD USE32 PUBLIC 'CODE'
            .
            .
            .
_TEXT       ENDS
```

5.2.2.3 Defining Segment Combinations with Combine Type

The optional *combine* type defines how to combine segments having the same name. The combine type can be any one of the following:

Combine Type	Meaning
PUBLIC	Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the segment.
STACK	Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the **PUBLIC** combine type, except that all addresses in the new segment are relative to the **SS** segment register. The stack pointer (**SP**) register is initialized to the length of the segment. The stack segment of your program should normally use the **STACK** type, since this automatically initializes the **SS** register, as described in Section 5.5.3. If you create a stack segment and do not use the **STACK** type, you must give instructions to initialize the **SS** and **SP** registers.
COMMON	Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address. If variables are initialized in more than one segment having the same name and **COMMON** type, the most recently initialized data replace any previously initialized data.
MEMORY	Concatenates all segments having the same name to form a single, contiguous segment. The Microsoft Overlay Linker treats **MEMORY** segments exactly the same as **PUBLIC** segments. **MASM** allows you to use **MEMORY** type even though **LINK** does not recognize a separate **MEMORY** type. This feature is compatible with other linkers that

may support a combine type conforming to the Intel definition of **MEMORY** type.

AT *address*

Causes all label and variable addresses defined in the segment to be relative to *address*.

The *address* can be any valid expression, but must not contain a forward reference—that is, a reference to a symbol defined later in the source file. An **AT** segment typically contains no code or initialized data. Instead, it represents an address template that can be placed over code or data already in memory, such as a screen buffer or other absolute memory locations defined by hardware. The linker will not generate any code or data for **AT** segments, but existing code or data can be accessed by name if it is given a label in an **AT** segment. Section 6.4, "Setting the Location Counter," shows an example of a segment with **AT** combine type.

The **AT** combine type has no meaning in protected-mode programs, since the segment represents a movable selector rather than a physical address. Real-mode programs that use **AT** segments must be modified before they can be used in protected mode. The planned multitasking version of DOS, OS/2, will provide DOS calls for doing tasks that are often done by manipulating memory directly under current versions of DOS.

If no *combine* type is given, the segment has private type. Segments having the same name are not combined. Instead, each segment receives its own physical segment when loaded into memory.

Notes

Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict. If types are given for an initial segment definition, then subsequent definitions for that segment need not specify any types.

Normally you should provide at least one stack segment (having **STACK** combine type) in a program. If no stack segment is declared, **LINK** displays a warning message. You can ignore this message if you

have a specific reason for not declaring a stack segment. For example, you would not have a separate stack segment in a program in the **.COM** format.

■ Example

The following source-code shell illustrates one way in which the *combine* and *align* types can be used. Figure 5.1 shows the way **LINK** would load the sample program into memory.

```
            NAME module_1

ASEG        SEGMENT WORD PUBLIC 'CODE'
start:      .
            .
            .
ASEG        ENDS

BSEG        SEGMENT WORD COMMON 'DATA'
            .
            .
            .
BSEG        ENDS

BSEG        SEGMENT PARA STACK 'STACK'
            .
            .
            .
CSEG        ENDS

DSEG        SEGMENT AT 0B800H
            .
            .
            .
DSEG        ENDS
            END start

            NAME module_2

ASEG        SEGMENT WORD PUBLIC 'CODE'
            .
            .
            .
ASEG        ENDS

BSEG        SEGMENT WORD COMMON 'DATA'
            .
            .
            .
BSEG        ENDS
```

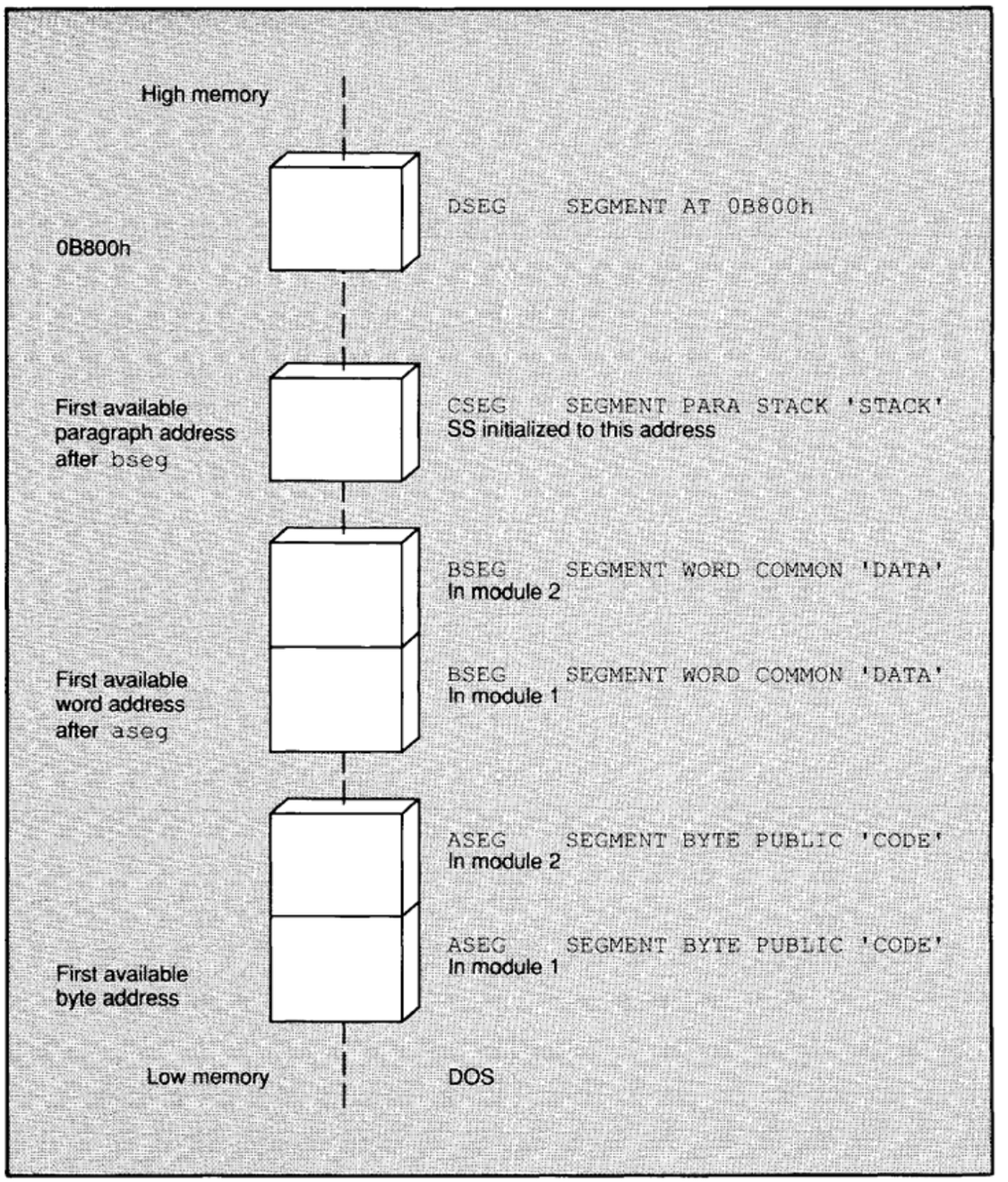


Figure 5.1 Segment Structure with Combine and Align Types

5.2.2.4 Controlling Segment Structure with Class Type

Class type is a means of associating segments that have different names, but similar purposes. It can be used to control segment order and to identify the code segment.

The *class* name must be enclosed in single quotation marks ('). Class names are not case sensitive unless the **/ML** or **/MX** option is used during assembly.

All segments belong to a class. Segments for which no class name is explicitly stated have the null class name. **LINK** imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64K.

Note

The names assigned for class types of segments should not be used for other symbol definitions in the source file. For example, if you give a segment the class name `'CONSTANT'`, you should not give the name `constant` to variables or labels in the source file.

The linker expects segments having the class name **CODE** or a class name with the suffix **CODE** to contain program code. You should always assign this class name to segments containing code.

The CodeView debugger also expects code segments to have the class name **CODE**. If you fail to assign a class type to a code segment, or if you give it a class type other than **CODE**, then labels may not be properly aligned for symbolic debugging.

Class type is one of two factors that control the final order of segments in an executable file. The other factor is the order of the segments in the source file (with the **/S** option or the **.SEQ** directive) or the alphabetical order of segments (with the **/A** option or the **.ALPHA** directive).

These factors control different internal behavior, but both affect final order of segments in the executable file. The sequential or alphabetical order of segments in the source file determines the order in which the assembler writes segments to the object file. The class type can affect the order in which the linker writes segments from object files to the executable file.

Segments having the same class type are loaded into memory together, regardless of their sequential or alphabetical order in the source file.

Note

The **DOSSEG** directive (see Section 5.1.2, "Specifying DOS Segment Order") overrides all other factors in determining segment order.

■ Example

```
A_SEG   SEGMENT 'SEG_1'
A_SEG   ENDS

B_SEG   SEGMENT 'SEG_2'
B_SEG   ENDS

C_SEG   SEGMENT 'SEG_1'
C_SEG   ENDS
```

When **MASM** assembles the preceding program fragment, it writes the segments to the object file in sequential or alphabetical order, depending on whether the **/A** option or the **.ALPHA** directive was used. In the example above, the sequential and alphabetical order are the same, so the order will be A_SEG, B_SEG, C_SEG in either case.

When the linker writes the segments to the executable file, it first checks to see if any segments have the same class type. If they do, it writes them to the executable file together. Thus A_SEG and C_SEG are placed together because they both have class type 'SEG_1'. The final order in memory is A_SEG, C_SEG, B_SEG.

Since **LINK** processes modules in the order it receives them on the command line, you may not always be able to easily specify the order you want segments to be loaded. For example, assume your program has four segments that you want loaded in the following order: _TEXT, _DATA, CONST, and STACK.

The _TEXT, CONST, and STACK segments are defined in the first module of your program, but the _DATA segment is defined in the second module. **LINK** will not put the segments in the proper order because it first loads the segments encountered in the first module.

You can avoid this problem by starting your program with dummy segment definitions in the order you wish to load your real segments. The dummy segments can either go at the start of the first module, or they can be placed in a separate include file that is called at the start of the first

module. You can then put the actual segment definitions in any order or any module you find convenient.

For example, you might call the following include file at the start of the first module of your program:

```
_TEXT       SEGMENT WORD PUBLIC 'CODE'
_TEXT       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
_DATA       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
CONST       ENDS
STACK       SEGMENT PARA STACK 'STACK'
STACK       ENDS
```

The **DOSSEG** directive may be more convenient for defining segment order if you are willing to accept the **DOS** segment-order conventions.

Once a segment has been defined, you do not need to specify the align, combine, use, and class types on subsequent definitions. For example, if your code defined dummy segments as shown above, you could define an actual data segment with the following statements:

```
_DATA       SEGMENT
            .
            .
            .
_DATA       ENDS
```

5.3 Defining Segment Groups

A group is a collection of segments associated with the same starting address. You may wish to use a group if you want several types of data to be organized in separate segments in your source code, but want them all to be accessible from a single, common segment register at run time.

■ Syntax

name **GROUP** *segment* [[,*segment*]]...

The *name* is the symbol assigned to the starting address of the group. All labels and variables defined within the segments of the group are relative to the start of the group, rather than to the start of the segments in which they are defined.

The *segment* can be any previously defined segment or a **SEG** expression (see Section 9.2.4.5).

Segments can be added to a group one at a time. For example, you can define and add segments to a group one by one. This is a new feature of Version 5.0. Previous versions required that all segments in a group be defined at one time.

The **GROUP** directive does not affect the order in which segments of a group are loaded. Loading order depends on each segment's class, or on the order in which object modules are given to the linker.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65,535 bytes.

Note

When the **MODEL** directive is used, the offset of a group-relative segment refers to the ending address of the segment, not the beginning. For example, the expression `OFFSET STACK` evaluates to the end of the stack segment.

Group names can be used with the **ASSUME** directive (discussed in Section 5.4, "Associating Segments with Registers") and as an operand prefix with the segment-override operator (discussed in Section 9.2.3).

■ Example

```
DGROUP      GROUP   ASEG,CSEG
            ASSUME  ds:DGROUP

ASEG        SEGMENT WORD PUBLIC 'DATA'
            .
asym        .
            .
ASEG        ENDS

BSEG        SEGMENT WORD PUBLIC 'DATA'
            .
bsym        .
            .
BSEG        ENDS

CSEG        SEGMENT WORD PUBLIC 'DATA'
            .
csym        .
            .
CSEG        ENDS
            END
```

Figure 5.2 shows the order of the example segments in memory. They are loaded in the order in which they appear in the source code (or in alphabetical order if the **.ALPHA** directive or **/A** option is specified).

Since ASEG and CSEG are declared part of the same group, they have the same base despite their separation in memory. This means that the symbols asym and csym have offsets from the beginning of the group, which is also the beginning of ASEG. The offset of bsym is from the beginning of BSEG, since it is not part of the group. This sample illustrates the way **LINK** organizes segments in a group. It is not intended as a typical use of a group.

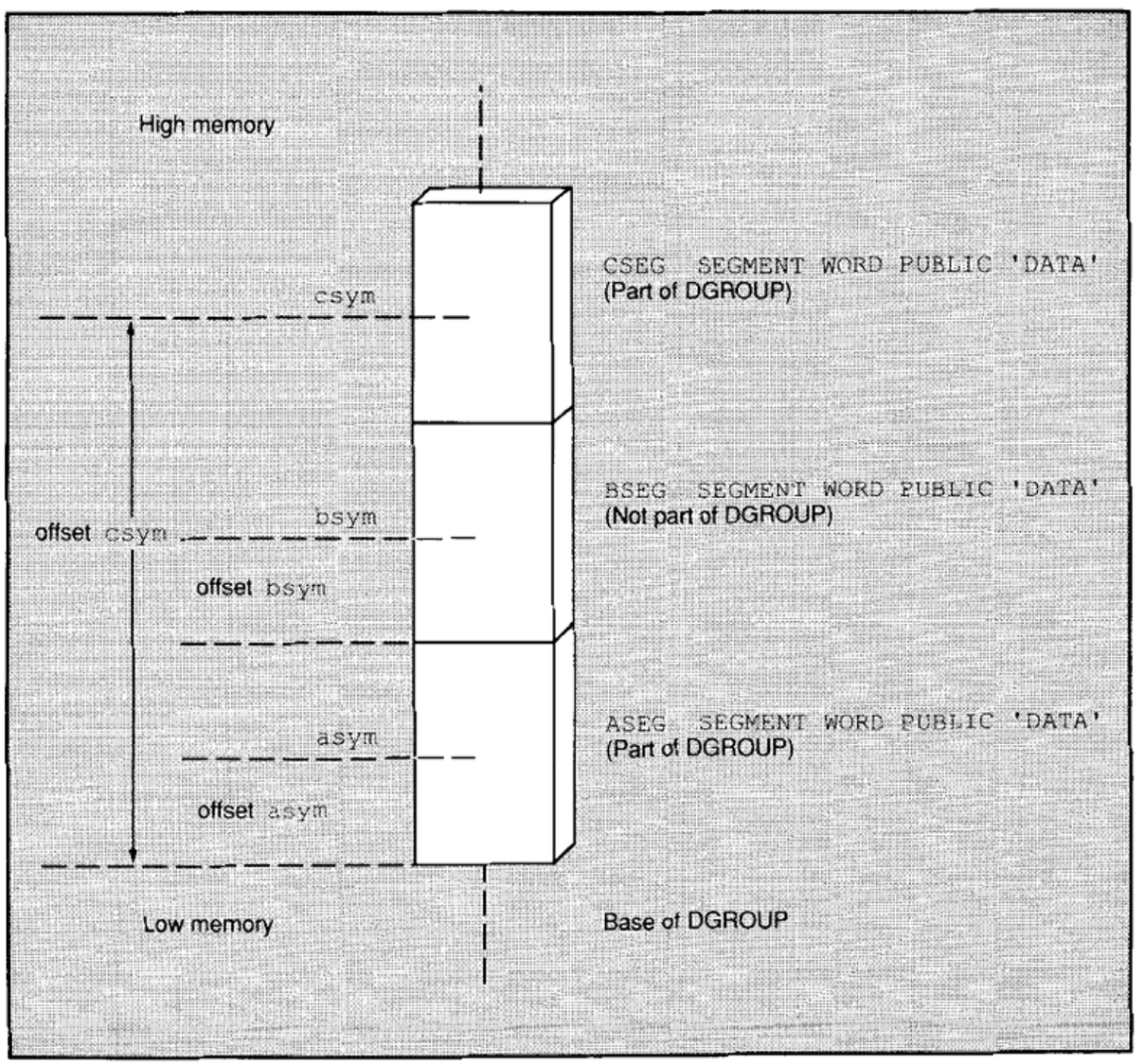


Figure 5.2 Segment Structure with Groups

5.4 Associating Segments with Registers

Many instructions assume a default segment. For example, **JMP** instructions assume the segment associated with the **CS** register; **PUSH** and **POP** instructions assume the segment associated with the **SS** register; **MOV** instructions assume the segment associated with the **DS** register.

When the assembler needs to reference an address, it must know what segment the address is in. It does this by using default segment or group addresses assigned with the **ASSUME** directive.

Note

Using the **ASSUME** directive to tell the assembler which segment to associate with a segment register is not the same as telling the processor. The **ASSUME** directive only affects assembly-time assumptions. You may need to use instructions to change run-time assumptions. Initializing segment registers at run time is discussed in Section 5.5.

■ Syntax

ASSUME *segmentregister*:*name* [[,*segmentregister*:*name*]]...
ASSUME *segmentregister*:**NOTHING**
ASSUME NOTHING

The *name* must be the name of the segment or group that is to be associated with the *segmentregister*. Subsequent instructions that assume a default register for referencing labels or variables automatically assume that if the default segment is *segmentregister*, then the label or variable is in the *name* segment or group.

The **ASSUME** directive can define a segment for each of the segment registers. The *segmentregister* can be **CS**, **DS**, **ES**, or **SS** (**FS** and **GS** are also available on the 80386). The *name* must be one of the following:

- The name of a segment defined in the source file with the **SEGMENT** directive
- The name of a group defined in the source file with the **GROUP** directive
- The keyword **NOTHING**

- A **SEG** expression (see Section 9.2.4.5, "SEG Operator")
- A string equate that evaluates to a segment or group name (but not a string equate that evaluates to a **SEG** expression)

The keyword **NOTHING** cancels the current segment selection. For example, the statement **ASSUME NOTHING** cancels all register selections made by previous **ASSUME** statements.

Usually a single **ASSUME** statement defines all four segment registers at the start of the source file. However, you can use the **ASSUME** directive at any point to change segment assumptions.

Using the **ASSUME** directive to change segment assumptions is often equivalent to changing assumptions with the segment-override operator (:) (see Section 9.2.3). The segment-override operator is more convenient for one-time overrides, whereas the **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

■ Example

```
                DOSSEG
                .MODEL  large       ; DS automatically assumed to @data
                .STACK  100h
                .DATA
d1              DW      7
                .FARDATA
d2              DW      9

                .CODE
start:          mov     ax,@data    ; Initialize near data
                mov     ds,ax
                mov     ax,@fardata ; Initialize far data
                mov     es,ax
                .
                .
                .

; Method 1 for series of instructions that need override
; Use segment override for each statement

                mov     ax,es:d2
                .
                .
                .
                mov     es:d2,bx

; Method 2 for series of instructions that need override
; Use ASSUME at beginning of series of instructions
                ASSUME  es:@fardata
                mov     cx,d2
                .
                .
                .
                mov     d2,dx
```

5.5 Initializing Segment Registers

Assembly-language programs must initialize segment values for each segment register before instructions that reference the segment register can be used in the source program.

Initializing segment registers is different from assigning default values for segment registers with the **ASSUME** statement. The **ASSUME** directive tells the assembler what segments to use at assembly time. Initializing segments gives them an initial value that will be used at run time.

Each of the segment registers is initialized in a different way.

5.5.1 Initializing the CS and IP Registers

The **CS** and **IP** registers are initialized by specifying a starting address with the **END** directive.

- **Syntax**

END [[*startaddress*]]

The *startaddress* is a label or expression identifying the address where you want execution to begin when the program is loaded. Normally a label for the *startaddress* should be placed at the address of the first instruction in the code segment.

The **CS** segment is initialized to the value of *startaddress*. The **IP** register is normally initialized to 0. You can change the initial value of the **IP** register by using the **ORG** directive (see Section 6.4, "Setting the Location Counter") just before the *startaddress* label. For example, programs in the **.COM** format use **ORG 100h** to initialize the **IP** register to 256 (100 hexadecimal).

If a program consists of a single source module, then the *startaddress* is required for that module. If a program has several modules, all modules must terminate with an **END** directive, but only one of them can define a *startaddress*.

Warning

One, and only one, module must define a *startaddress*. If you do not specify a *startaddress*, none is assumed. Neither **MASM** nor **LINK** will generate an error message, but your program will probably start execution at the wrong address.

■ Example

```
; Module 1
                .CODE
start:          .                       ; First executable instruction
                .
                .
                EXTRN   task:NEAR
                call    task
                .
                .
                .
                END     start           ; Starting address defined in main module

; Module 2
                PUBLIC  task
                .CODE
task            PROC
                .
                .
                .
task            ENDP
                END                     ; No starting address in secondary module
```

If `Module 1` and `Module 2` are linked into a single program, it is essential that only the calling module define a starting address.

5.5.2 Initializing the DS Register

The **DS** register must be initialized to the address of the segment that will be used for data.

The address of the segment or group for the initial data segment must be loaded into the **DS** register. This is done in two statements because a

memory value cannot be loaded directly into a segment register. The segment-setup lines typically appear at the start or very near the start of the code segment.

■ Example 1

```
_DATA       SEGMENT WORD PUBLIC 'DATA'
            .
            .
            .
_DATA       ENDS
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            ASSUME  cs:_TEXT,ds:_DATA
start:      mov     ax,_DATA            ; Load start of data segment
            mov     ds,ax               ; Transfer to DS register
            .
            .
            .
_TEXT       ENDS
            END     start
```

If you are using the Microsoft naming convention and segment order, the address loaded into the **DS** register is not a segment address but the address of **DGROUP**, as shown in Example 2. With simplified segment directives, the address of **DGROUP** is represented by the predefined equate **@data**.

■ Example 2

```
            DOSSEG
            .MODEL  SMALL
            .DATA
            .
            .
            .
            .CODE
start:      mov     ax,@data            ; Load start of DGROUP (@data)
            mov     ds,ax               ; Transfer to DS register
            .
            .
            .
            END     start
```

5.5.3 Initializing the SS and SP Registers

The SS register is automatically initialized to the value of the last segment in the source code having combine type **STACK**. The SP register is automatically initialized to the size of the stack segment. Thus SS:SP initially points to the end of the stack.

If you use a stack segment with combine type **STACK**, initialization of SS and SP is automatic. The stack is automatically set up in this way with the simplified segment directives.

However, you can initialize or reinitialize the stack segment directly by changing the values of SS and SP. Since hardware interrupts use the same stack as the program, you should turn off hardware interrupts while changing the stack. Most 8086-family processors do this automatically, but early versions of the 8088 do not.

■ Example

```
            .MODEL  small
            .STACK  100h                ; Initialize "STACK"
            .DATA
            .
            .
            .
            .CODE
start:      mov     ax,@data            ; Load segment location
            mov     ds,ax               ;   into DS register
            cli                         ; Turn off interrupts
            mov     ss,ax               ; Load same value as DS into SS
            mov     sp,OFFSET STACK     ; Give SP new stack size
            sti                         ; Turn interrupts back on
            .
            .
            .
```

This example reinitializes SS so that it has the same value as DS, and adjusts SP to reflect the new stack offset. Microsoft high-level-language compilers do this so that stack variables in near procedures can be accessed relative to either SS or DS.

5.5.4 Initializing the ES Register

The **ES** register is not automatically initialized. If your program uses the **ES** register, you must initialize it by moving the appropriate segment value into the register.

■ **Example**

```
ASSUME  es:@fardata         ; Tell the assembler
mov     ax,@fardata         ; Tell the processor
mov     es,ax
```

5.6 Nesting Segments

Segments can be nested. When **MASM** encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, **MASM** continues assembly of the enclosing segment.

Nesting of segments makes it possible to mix segment definitions in programs that use simplified segment directives for most segment definitions. When a full segment definition is given, the new segment is nested in the simplified segment in which it is defined.

■ **Example 1**

```
; Macro to print message on the screen
; Uses full segment definitions - segments nested

message     MACRO   text
            LOCAL   symbol
_DATA       SEGMENT WORD PUBLIC 'DATA'
symbol      DB      &text
            DB      13,10,"$"
_DATA       ENDS
            mov     ah,09h
            mov     dx,OFFSET symbol
            int     21h
            ENDM

_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            .
            .
            .
            message "Please insert disk"
```

In the example above, a macro called from inside of the code segment (_TEXT) allocates a variable within a nested data segment (_DATA). This has the effect of allocating more data space on the end of the data segment each time the macro is called. The macro can be used for messages appearing only once in the source code.

■ Example 2

```
; Macro to print message on the screen
; Uses simplified segment directives - segments not nested

message     MACRO   text
            LOCAL   symbol
            .DATA
symbol      DB      &text
            DB      13,10,"$"
            .CODE
            mov     ah,09h
            mov     dx,OFFSET symbol
            int     21h
            ENDM

            .CODE
            .
            .
            .
            message "Please insert disk"
```

Although Example 2 has the same practical effect as Example 1, MASM handles the two macros differently. In Example 1, assembly of the outer (code) segment is suspended rather than terminated. In Example 2, assembly of the code segment terminates, assembly of the data segment starts and terminates, and then assembly of the code segment is restarted.

CHAPTER 6

DEFINING LABELS AND VARIABLES

6.1 Using Type Specifiers 119

6.2 Defining Code Labels 120
- 6.2.1 Near Code Labels 120
- 6.2.2 Procedure Labels 121
- 6.2.3 Code Labels Defined with the LABEL Directive 122

6.3 Defining and Initializing Data 123
- 6.3.1 Variables 123
  - 6.3.1.1 Integer Variables 124
  - 6.3.1.2 Binary Coded Decimal Variables 127
  - 6.3.1.3 String Variables 127
  - 6.3.1.4 Pointer Variables 128
  - 6.3.1.5 Real-Number Variables 130
- 6.3.2 Arrays and Buffers 135
- 6.3.3 Labeling Variables 136

6.4 Setting the Location Counter 137

6.5 Aligning Data 138

This chapter explains how to define labels, variables, and other symbols that refer to instruction and data locations within segments.

The label- and variable-definition directives described in this chapter are closely related to the segment-definition directives described in Chapter 5, "Defining Segment Structure." Segment directives assign the addresses for segments. The variable-and label-definition directives assign offset addresses within segments.

The assembler assigns offset addresses for each segment by keeping track of a value called the location counter. The location counter is incremented as each source statement is processed so that it always contains the offset of the location being assembled. When a label or a variable name is encountered, the current value of the location counter is assigned to the symbol.

This chapter tells you how to assign labels and most kinds of variables. (Multifield variables such as structures and records are discussed in Chapter 7, "Using Structures and Records.") The chapter also discusses related directives, including those that control the location counter directly.

6.1 Using Type Specifiers

Some statements require type specifiers to give the size or type of an operand. There are two kinds of type specifiers: those that specify the size of a variable or other memory operand, and those that specify the distance of a label.

The type specifiers that give the size of a memory operand are listed below with the number of bytes specified by each:

Specifier	Number of Bytes
BYTE	1
WORD	2
DWORD	4
FWORD	6
QWORD	8
TBYTE	10

In some contexts, **ABS** can also be used as a type specifier that indicates an operand is a constant rather than a memory operand.

The type specifiers that give the distance of a label are listed below:

Specifier	Description
FAR	The label references both the segment and offset of the label.
NEAR	The label references only the offset of the label.
PROC	The label has the default type (near or far) of the current memory model. The default size is always near if you use full segment definitions. If you use simplified segment definitions (see Section 5.1) the default type is near for small and compact models or far for medium, large, and huge models.

Directives that use type specifiers include **LABEL**, **PROC**, **EXTRN**, and **COMM**. Operators that use type specifiers include **PTR** and **THIS**.

6.2 Defining Code Labels

Code labels give symbolic names to the addresses of instructions in the code segment. These labels can be used as the operands to jump, call, and loop instructions to transfer program control to a new instruction. There are three types of code labels: near labels, procedure labels, and labels created with the **LABEL** directive.

6.2.1 Near Code Labels

Near-label definitions create instruction labels that have **NEAR** type. These instruction labels can be used to access the address of the label from other statements.

- **Syntax**

name:

The *name* must not be previously defined in the module and it must be followed by a colon (:). Furthermore, the segment containing the definition must be the one that the assembler currently associates with the **CS** register. The **ASSUME** directive is used to associate a segment with a segment register (see Section 5.4, "Associating Segments with Registers").

A near label can appear on a line by itself or on a line with an instruction. The same label name can be used in different modules as long as each label is only referenced by instructions in its own module. If a label must be referenced by instructions in another module, it must be given a unique name and declared with the **PUBLIC** and **EXTRN** directives, as described in Chapter 8, "Creating Programs from Multiple Modules."

■ Examples

```
            cmp     ax,5        ; Compare with 5
            ja      bigger
            jb      smaller
            .                   ; Instructions if AX = 5
            .
            .
            jmp     done
bigger:     .                   ; Instructions if AX > 5
            .
            .
            jmp     done
smaller:    .                   ; Instructions if AX < 5
            .
            .
done:
```

6.2.2 Procedure Labels

The start of an assembly-language procedure can be defined with the **PROC** directive, and the end of the procedure can be defined with the **ENDP** directive.

■ Syntax

label **PROC** [[**NEAR**|**FAR**]]
statements
RET [[*constant*]]
`label` **ENDP**

The `label` assigns a symbol to the procedure. The distance can be **NEAR** or **FAR**. Any **RET** instructions within the procedure automatically have the same distance (**NEAR** or **FAR**) as the procedure. Procedures and the **RET** instruction are discussed in more detail in Section 17.4, "Using Procedures."

The **ENDP** directive labels the address where the procedure ends. Every procedure label must have a matching **ENDP** label to mark the end of the procedure. **MASM** generates an error message if it does not find an **ENDP** directive to match each **PROC** directive.

When the **PROC** label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to **NEAR** or **FAR**. If the label has **FAR** type, the assembler also sets its segment value to that of the enclosing segment. If you have specified full segment definitions, the default distance is **NEAR**. If you are using simplified segment definitions, the default distance is the distance associated with the declared memory model—that is, **NEAR** for small and compact models or **FAR** for medium, large, and huge models.

The procedure label can be used in a **CALL** instruction to direct execution control to the first instruction of the procedure. Control can be transferred to a **NEAR** procedure label from any address in the same segment as the label. Control can be transferred to a **FAR** procedure label from an address in any segment.

Procedure labels must be declared with the **PUBLIC** and **EXTRN** directives if they are located in one module but called from another module, as described in Chapter 8, "Creating Programs from Multiple Modules."

■ Examples

```
            call    task        ; Call procedure
            .
            .
            .
task        PROC    NEAR        ; Start of procedure
            .
            .
            .
            ret
task        ENDP                ; End of procedure
```

6.2.3 Code Labels Defined with the LABEL Directive

The **LABEL** directive provides an alternative method of defining code labels.

■ Syntax

name **LABEL** *distance*

The *name* is the symbol name assigned to the label. The *distance* can be a type specifier such as **NEAR**, **FAR**, or **PROC**. **PROC** means **NEAR** or **FAR**, depending on the default memory model, as described in Section 4.4, "Starting and Ending Source Files." You can use the **LABEL** directive to define a second entry point into a procedure. **FAR** code labels can also be the destination of far jumps or of far calls that use the **RETF** instruction (see Section 17.4.2, "Defining Procedures").

■ **Example**

```
task        PROC   FAR          ; Main entry point
            .
            .
            .
task1       LABEL  FAR          ; Secondary entry point
            .
            .
            .
            ret
task        ENDP                ; End of procedure
```

6.3 Defining and Initializing Data

The data-definition directives enable you to allocate memory for data. At the same time, you can specify the initial values for the allocated data. Data can be specified as numbers, strings, or expressions that evaluate to constants. The assembler translates these constant values into binary bytes, words, or other units of data. The encoded data are written to the object file at assembly time.

6.3.1 Variables

Variables consist of one or more named data objects of a specified size.

■ **Syntax**

[[*name*]] *directive initializer* [[,*initializer*]]...

The *name* is the symbol name assigned to the variable. If no *name* is assigned, the data is allocated; but the starting address of the variable has no symbolic name.

The size of the variable is determined by *directive*. The directives that can be used to define single-item data objects are listed below:

Directive	Meaning
DB	Defines byte
DW	Defines word (2 bytes)
DD	Defines doubleword (4 bytes)
DF	Defines farword (6 bytes); normally used only with 80386 processor

DQ	Defines quadword (8 bytes)
DT	Defines 10-byte variable

The optional *initializer* can be a constant, an expression that evaluates to a constant, or a question mark (**?**). The question mark is the symbol indicating that the value of the variable is undefined. You can define multiple values by using multiple initializers separated by commas, or by using the **DUP** operator, as explained in Section 6.3.2, "Arrays and Buffers."

Simple data types can allocate memory for integers, strings, addresses, or real numbers.

6.3.1.1 Integer Variables

When defining an integer variable, you can specify an initial value as an integer constant or as a constant expression. **MASM** generates an error if you specify an initial value too large for the specified variable.

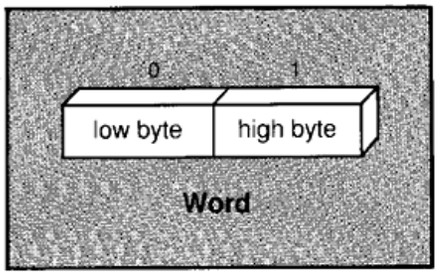
Integer values for all sizes except 10-byte variables are stored in the complement format of the binary two. They can be interpreted as either signed or unsigned numbers. For instance, the hexadecimal value 0FFCD can be interpreted either as the signed number –51 or the unsigned number 65,485.

The processor cannot tell the difference between signed and unsigned numbers. Some instructions are designed specifically for signed numbers. It is the programmer's responsibility to decide whether a value is to be interpreted as signed or unsigned, and then to use the appropriate instructions to handle the value correctly.

The directives for defining integer variables are listed below with the sizes of integer they can define:

Directive	Size
DB (bytes)	Allocates unsigned numbers from 0 to 255 or signed numbers from –128 to 127. These values can be used directly in 8086-family instructions.
DW (words)	Allocates unsigned numbers from 0 to 65,535 or signed numbers from –32,768 to 32,767. The

bytes of a word integer are stored in the format shown below:

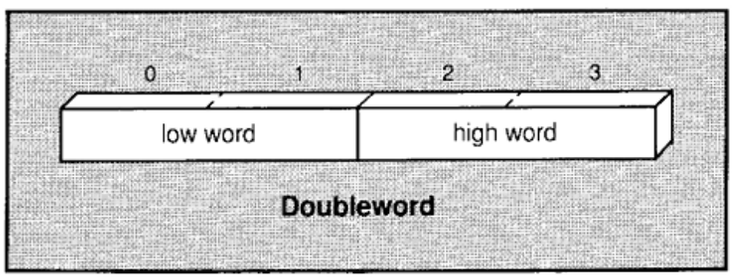


Note that in assembler listings and in most debuggers (including the CodeView debugger) the bytes of a word are shown in the opposite order—high byte first—since this is the way most people think of numbers. For instance, the decimal value 1987 is shown as 07C3h in listings and with the Dump Words (**DW**) CodeView command. Internally, the number is stored as C307h.

Word values can be used directly in 8086-family instructions. They can also be loaded, used in calculations, and stored with 8087-family instructions.

DD (doublewords)

Allocates unsigned numbers from 0 to 4,294,967,295 or signed numbers from -2,147,483,648 to 2,147,483,647. The words of a doubleword integer are stored in the format shown below:



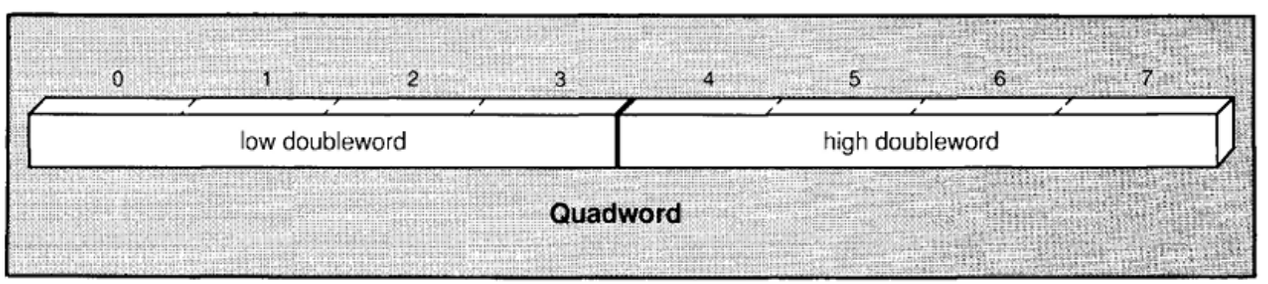
These 32-bit values (called long integers) can be loaded, used in calculations, and stored with

8087-family instructions. Some calculations can be done on these numbers directly with 16-bit 8086-family processors; others involve an indirect method of doing calculations on each word separately (see Section 16.1, "Adding"). These long integers can be used directly in calculations with the 80386 processor.

DF (farwords) Allocates 6-byte (48-bit) integers.

These values are normally only used as pointer variables on the 80386 processor (see Section 6.2.1.4).

DQ (quadwords) Allocates 64-bit integers. The doublewords of a quadword integer are stored in the format shown below:



These values can be loaded, used in calculations, and stored with 8087-family instructions. You must write your own routines to use them with 16-bit 8086-family processors. Some calculations can be done on these numbers directly with the 80386 processor, but others require an indirect method of doing calculations on each doubleword separately (see Section 16.1, "Adding").

DT Allocates 10-byte (80-bit) integers if the **D** radix specifier is used.

By default, **DT** allocates packed BCD (binary coded decimal) numbers, as described in Section 6.3.1.2, "Binary Coded Decimal Variables." If you define binary 10-byte integers, you must write your own routines to use routines in calculations.

■ **Example**

```
integer     DB      16          ; Initialize byte to 16
expression  DW      4*3         ; Initialize word to 12
empty       DQ      ?           ; Allocate uninitialized long integer
            DB      1,2,3,4,5,6 ; Initialize six unnamed bytes
high_byte   DD      4294967295  ; Initialize double word to 4,294,967,295
tb          DT      2345d       ; Initialize 10-byte binary integer
```

6.3.1.2 Binary Coded Decimal Variables

Binary coded decimals (BCD) provide a method of doing calculations on large numbers without rounding errors. They are sometimes used in financial applications. There are two kinds: packed and unpacked.

Unpacked BCD numbers are stored one digit to a byte, with the value in the lower four bits. They can be defined with the **DB** directive. For example, an unpacked BCD number could be defined and initialized as shown below:

```
unpackedr   DB      1,5,8,2,5,2,9   ; Initialized to 9,252,851
unpackedf   DB      9,2,5,2,8,5,1   ; Initialized to 9,252,851
```

Whether least-significant digits can come either first or last, depends on how you write the calculation routines that handle the numbers. Calculations with unpacked BCD numbers are discussed in Section 16.5.1.

Packed BCD numbers are stored two digits to a byte, with one digit in the lower four bits and one in the upper four bits. The leftmost bit holds the sign (0 for positive or 1 for negative).

Packed BCD variables can be defined with the **DT** directive as shown below:

```
packed      DT      9252851         ; Allocate 9,252,851
```

The 8087-family processors can do fast calculations with packed BCD numbers, as described in Chapter 19, "Calculating with a Math Coprocessor." The 8086-family processors can also do some calculations with packed BCD numbers, but the process is slower and more complicated. See Section 16.5.2 for details.

6.3.1.3 String Variables

Strings are normally initialized with the **DB** directive. The initializing value is specified as a string constant. Strings can also be initialized by specifying each value in the string. For example, the following definitions are equivalent:

```
version1    DB      97,98,99            ; As ASCII values
version2    DB      'a','b','c'         ; As characters
version3    DB      "abc"               ; As a string
```

One- and two-character strings (four-character strings on the 80386) can also be initialized with any of the other data-definition directives. The last (or only) character in the string is placed in the byte with the lowest address. Either 0 or the first character is placed in the next byte. The unused portion of such variables is filled with zeros.

■ Examples

```
function9   DB      'Hello',13,10,'$'   ; Use with DOS INT 21h
                                        ;   function 9

asciiz      DB      "\ASM\TEST.ASM",0   ; Use as ASCIIZ string

message     DB      "Enter file name: " ; Use with DOS INT 21h
l_message   EQU     $-message           ;   function 40h
a_message   EQU     OFFSET message

str1        DB      "ab"                ; Stored as 61 62
str2        DD      "ab"                ; Stored as 62 61 00 00
str3        DD      "a"                 ; Stored as 61 00 00 00
```

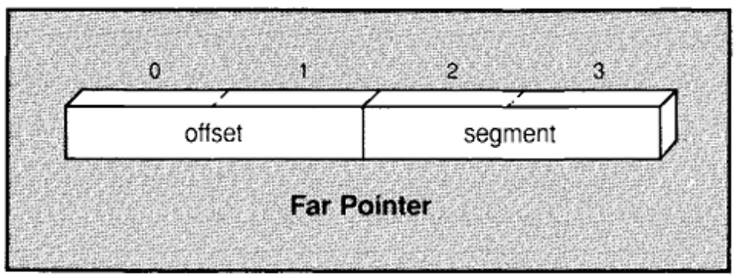
6.3.1.4 Pointer Variables

Pointer variables (or pointers) are variables that contain the address of a data or code object rather than the object itself. The address in the variable "points" to another address. Pointers can be either near addresses or far addresses.

Near pointers consist of the offset portion of the address. They can be initialized in word variables by using the **DW** directive. Values in near-address variables can be used in situations where the segment portion of the address is known to be the current segment.

Far pointers consist of both the segment and offset portions of the address. They can be initialized in doubleword variables, using the **DD** directive. Values in far-address variables must be used when the segment portion of

the address may be outside the current segment. The segment and offset of a far pointer are stored in the format shown below:

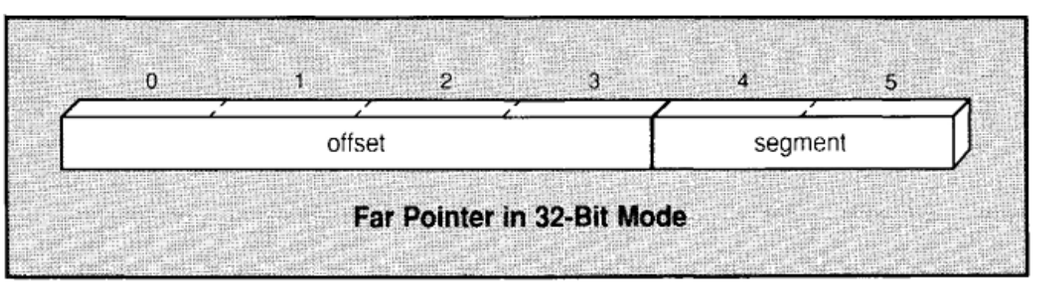


■ Examples

```
string      DB      "Text",0     ; Null-terminated string
npstring    DW      string       ; Near pointer to "string"
fpstring    DD      string       ; Far pointer to "string"
```

■ 80386 Only

Pointers are different on the 80386 processor if the **USE32** use type has been specified. In this case the offset portion of an address consists of 32 bits, and the segment portion consists of 16 bits. Therefore a near pointer is 32 bits (a doubleword), and a far pointer is 48 bits (a farword). The segment and offset of a 32-bit-mode far pointer are stored in the format shown below:



■ Example

```
_DATA       SEGMENT WORD USE32 PUBLIC 'DATA'
string      DB      "Text",0     ; Null-terminated string
npstring    DD      string       ; Near (32-bit) pointer to "string"
fpstring    DF      string       ; Far (48-bit) pointer to "string"
_DATA       ENDS
```

6.3.1.5 Real-Number Variables

Real numbers must be stored in binary format. However, when initializing variables, you can specify decimal or hexadecimal constants and let the assembler automatically encode them into their binary equivalents. **MASM** can use two different binary formats for real numbers: IEEE or Microsoft Binary. You can specify the format by using directives (IEEE is the default).

This section tells you how to initialize real-number variables, describes the two binary formats, and explains real-number encoding.

Initializing and Allocating Real-Number Variables

Real numbers can be defined by initializing them either with real-number constants or with encoded hexadecimal constants. The real-number designator (**R**) must follow numbers specified in encoded format.

The directives for defining real numbers are listed below with the sizes of the numbers they can allocate:

Directive	Size
DD	Allocates short (32-bit) real numbers in either the IEEE or Microsoft Binary format.
DQ	Allocates long (64-bit) real numbers in either the IEEE or Microsoft Binary format.
DT	Allocates temporary or 10-byte (80-bit) real numbers. The format of these numbers is similar to the IEEE format. They are always encoded the same regardless of the real-number format. Their size is nonstandard and incompatible with Microsoft high-level languages. Temporary-real format is provided for those who want to initialize real numbers in the format used internally by 8087-family processors.

The 8086-family microprocessors do not have any instructions for handling real numbers. You must write your own routines, use a library that includes real-number calculation routines, or use a coprocessor. The 8087-family coprocessors can load real numbers in the IEEE format; they can also use the values in calculations and store the results back to memory, as explained in Chapter 19, "Calculating with a Math Coprocessor."

■ Examples

```
shrt        DD      98.6                ; MASM automatically encodes
long        DQ      5.391E-4            ;   in current format
ten_byte    DT      -7.31E7

eshrt       DD      87453333r           ; 98.6 encoded in Microsoft
                                        ; Binary format
elong       DQ      3F41AA4C6F445B7Ar   ; 5.391E-4 encoded in IEEE format
```

The real-number designator (R) used to specify encoded numbers is explained in Section 4.3.2, "Packed Binary Coded Decimal Constants."

Selecting a Real-Number Format

MASM can encode four- and eight-byte real numbers in two different formats: IEEE and Microsoft Binary. Your choice depends on the type of program you are writing. The four primary alternatives are listed below:

1. If your program requires a coprocessor for calculations, you must use the IEEE format.

2. Most high-level languages use the IEEE format. If you are writing modules that will be called from such a language, your program should use the IEEE format. All versions of the C, FORTRAN, and Pascal compilers sold by Microsoft and IBM use the IEEE format.

3. If you are writing a module that will be called from most previous versions of Microsoft or IBM BASIC, your program should use the Microsoft Binary format. Versions that support only the Microsoft Binary format include:

   - Microsoft QuickBASIC through Version 2.01
   - Microsoft BASIC Compiler through Version 5.3
   - IBM BASIC Compiler through Version 2.0
   - Microsoft GW-BASIC interpreter (all versions)
   - IBM BASICA interpreter (all versions)

   Microsoft QuickBASIC Version 3.0 supports both the Microsoft Binary and IEEE formats as options.

   Future versions of Microsoft QuickBASIC and the BASIC compiler will support only the IEEE format.

4. If you are creating a stand-alone program that does not use a coprocessor, you can choose either format. The IEEE format is better for overall compatibility with high-level languages. Also, the CodeView debugger can display only real numbers in the IEEE format. The Microsoft Binary format may be necessary for compatibility with existing source code.

Note

When you interface assembly-language modules with high-level languages, the real-number format only matters if you initialize real-number variables in the assembly module. If your assembly module does not use real numbers, or if all real numbers are initialized in the high-level-language module, the real-number format does not make any difference.

By default, **MASM** assembles real-number data in the IEEE format. This is a change from previous versions of the assembler, which used the Microsoft Binary format by default. If you wish to use the Microsoft Binary format, you must put the **.MSFLOAT** directive at the start of your source file before initializing any real-number variables (see Section 4.5.1, "Starting and Ending Source Files").

Real-Number Encoding

The IEEE format for encoding four- and eight-byte real numbers is illustrated in Figure 6.1.

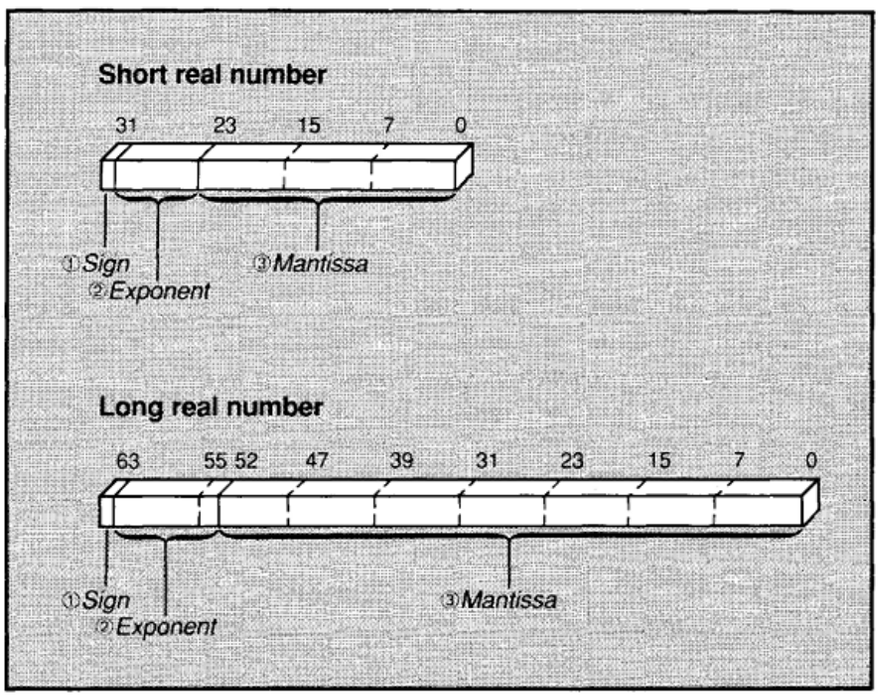


Figure 6.1 Encoding for Real Numbers in IEEE Format

The parts of the real numbers are described below:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (8 bits for short real number or 11 bits for long real number).
3. All except the first set bit of mantissa in the remaining bits of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short real numbers and 52 bits for long real numbers.

The Microsoft Binary format for encoding real numbers is illustrated in Figure 6.2.

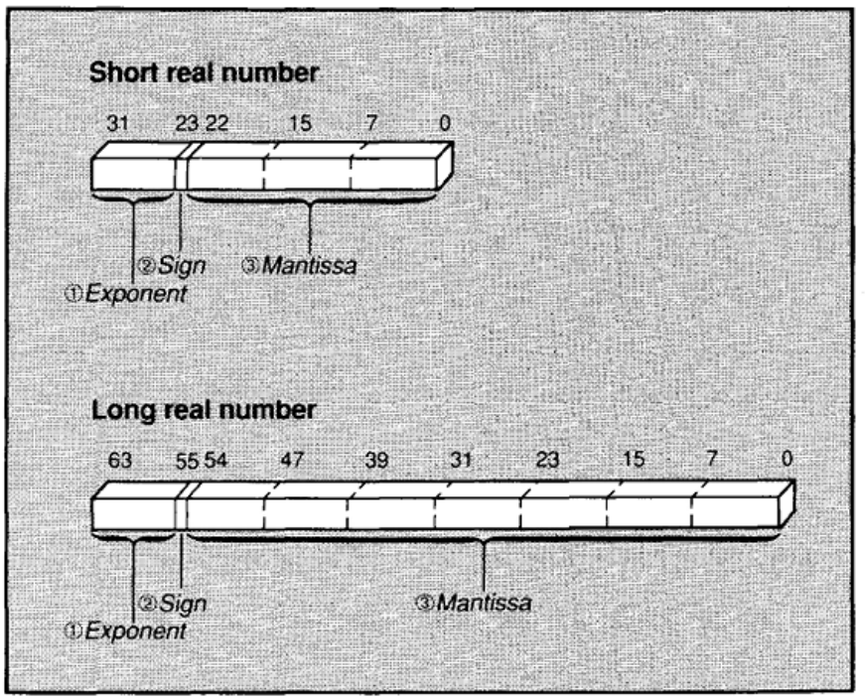


Figure 6.2 Encoding for Real Numbers in Microsoft Binary Format

The three parts of real numbers are described below:

1. Biased exponent (8 bits) in the high-address byte. The bias is 81h for short real numbers and 401h for long real numbers.
2. Sign bit (0 for positive or 1 for negative) in the upper bit of the second-highest byte.

3. All except the first set bit of mantissa in the remaining 7 bits of the second-highest byte and in the remaining bytes of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short real numbers and 55 bits for long real numbers.

MASM also supports the 10-byte temporary-real format used internally by 8087-family coprocessors. This format is similar to IEEE format. The size is nonstandard and is not used by Microsoft compilers or interpreters. Since the coprocessors can load and automatically convert numbers in the more standard 4- and 8-byte formats, the 10-byte format is seldom used in assembly-language programming.

The temporary-real format for encoding real numbers is illustrated in Figure 6.3.

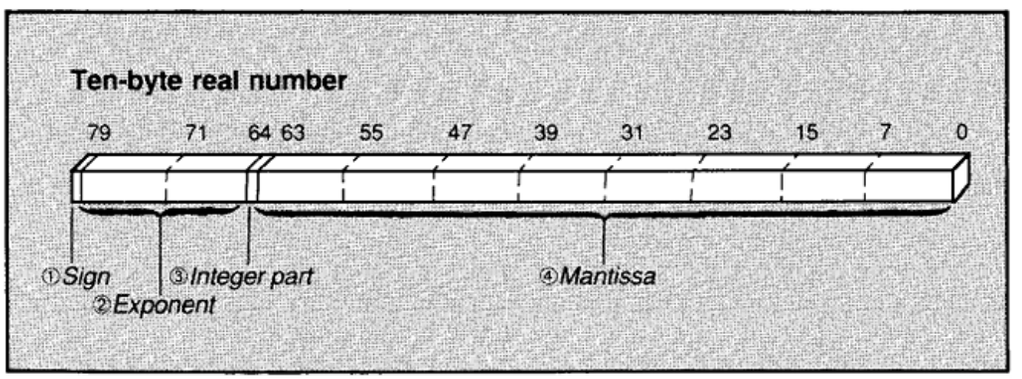


Figure 6.3 Encoding for Real Numbers in Temporary-Real Format

The four parts of the real numbers are described below:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (15 bits for 10-byte real).
3. The integer part of mantissa in the next bit in sequence (bit 63).
4. Remaining bits of mantissa in the remaining bits of the variable. The length is 63 bits.

Notice that the 10-byte temporary-real format stores the integer part of the mantissa. This differs from the 4- and 8-byte formats, in which the integer part is implicit.

6.3.2 Arrays and Buffers

Arrays, buffers, and other data structures consisting of multiple data objects of the same size can be defined with the **DUP** operator. This operator can be used with any of the data-definition directives described in this chapter.

■ Syntax

count **DUP** (*initialvalue*[[,*initialvalue*]]...)

The *count* sets the number of times to define *initialvalue*. The initial value can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. It can also be the undefined symbol (?) if there is no initial value.

Multiple initial values must be separated by commas. If multiple values are specified within the parentheses, the sequence of values is allocated *count* times. For example, the statement

```
DB      5 DUP ("Text ")
```

allocates the string "Text " five times for a total of 20 bytes.

DUP operators can be nested up to 17 levels. The initial value (or values) must always be placed within parentheses.

■ Examples

```
array       DD      10 DUP (1)                      ; 10 doublewords
                                                    ;   initialized to 1

buffer      DB      256 DUP (?)                     ; 256 byte buffer

masks       DB      20 DUP (040h,020h,04h,02h)      ; 80 byte buffer
                                                    ;   with bit masks
            DB      32 DUP ("I am here ")           ; 320 byte buffer with
                                                    ;   signature for debugging
three_d     DD      5 DUP (5 DUP (5 DUP (0)))       ; 125 doublewords
                                                    ;  initialized to 0
```

Note

MASM sometimes generates different object code when the **DUP** operator is used rather than when multiple values are given. For example, the statement

```
test1       DB      ?,?,?,?,?  ; Indeterminate
```

is "indeterminate." It causes **MASM** to write five zero-value bytes to the object file. The statement

```
test2       DB      5 DUP (?)  ; Undefined
```

is "undefined." It causes **MASM** to increase the offset of the next record in the object file by five bytes. Therefore an object file created with the first statement will be larger than one created with the second statement.

In most cases, the distinction between indeterminate and undefined definitions is trivial. The linker adjusts the offsets so that the same executable file is generated in either case. However, the difference is significant in segments with the **COMMON** combine type. If **COMMON** segments in two modules contain definitions for the same variable, one with an indeterminate value and one with an explicit value, the actual value in the executable file varies depending on link order. If the module with the indeterminate value is linked last, the 0 initialized for it overrides the explicit value. You can prevent this by always using undefined rather than indeterminate values in **COMMON** segments. For example, use the first of the following statements:

```
test3       DB      1 DUP (?)  ; Undefined - doesn't initialize
test4       DB      ?          ; Indeterminate - initializes 0
```

If you use the undefined definition, the explicit value is always used in the executable file regardless of link order.

6.3.3 Labeling Variables

The **LABEL** directive can be used to define a variable of a given size at a specified location. It is useful if you want to refer to the same data as variables of different sizes.

■ Syntax

name **LABEL** *type*

The *name* is the symbol assigned to the variable, and *type* is the variable size. The type can be any one of the following type specifiers: **BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**. It can also be the name of a previously defined structure.

■ Examples

```
warray      LABEL   WORD        ; Access array as 50 words
darray      LABEL   DWORD       ; Access same array as 25 doublewords
barray      DB      100 DUP(?)  ; Access same array as 100 bytes
```

6.4 Setting the Location Counter

The location counter is the value **MASM** maintains to keep track of the current location in the source file. The location counter is incremented automatically as each source statement is processed. However, the location counter can be set specifically using the **ORG** directive.

■ Syntax

ORG *expression*

Subsequent code and data offsets begin at the new offset specified set by *expression*. The *expression* must resolve to a constant number. In other words, all symbols used in the expression must be known on the first pass of the assembler.

Note

The value of the location counter, represented by the dollar sign ($), can be used in *expression*, as described in Section 9.3, "Using the Location Counter."

■ Example 1

```
; Labeling absolute addresses

STUFF       SEGMENT AT 0        ; Segment has constant value 0
            ORG     410h        ; Offset has constant value 410h
equipment   LABEL   WORD        ; Value at 0000:0410 labeled "equipment"
            ORG     417h        ; Offset has constant value 417h
keyboard    LABEL   WORD        ; Value at 0000:0417 labeled "keyboard"
STUFF       ENDS

            .CODE
            .
            .
            .
            ASSUME  ds:STUFF    ; Tell the assembler
            mov     ax,STUFF    ; Tell the processor
            mov     ds,ax

            mov     dx,equipment
            mov     keyboard,ax
```

Example 1 illustrates one way of assigning symbolic names to absolute addresses. This technique is not possible under protected-mode operating systems.

■ Example 2

```
; Format for .COM files

_TEXT       SEGMENT
            ASSUME cs:_TEXT,ds:_TEXT,ss:_TEXT,es:_TEXT
            ORG    100h         ; Skip 100h bytes of DOS header

entry:      jmp    begin        ; Jump over data
variable    DW     ?            ; Put more data here
            .
begin:      .                   ; First line of code
            .                   ; Put more code here
_TEXT       ENDS
            END    entry
```

Example 2 illustrates how the **ORG** directive is used to initialize the starting execution point in **.COM** files.

6.5 Aligning Data

Some operations are more efficient when the variable used in the operation is lined up on a boundary of a particular size. The **ALIGN** and **EVEN** directives can be used to pad the object file so that the next variable is aligned on a specified boundary.

■ Syntax 1

EVEN

■ Syntax 2

ALIGN *number*

The **EVEN** directive always aligns on the next even byte. The **ALIGN** directive aligns on the next byte that is a multiple of *number*. The *number* must be a power of 2. For example, use `ALIGN 2` or `EVEN` to align on word boundaries, or use `ALIGN 4` to align on doubleword boundaries.

If the value of the location counter is not on the specified boundary when an **ALIGN** directive is encountered, the location counter is incremented to a value on the boundary. **NOP** (no operation) instructions are generated to pad the object file. If the location counter is already on the boundary, the directive has no effect.

The **ALIGN** and **EVEN** directives give no efficiency improvements on processors that have an 8-bit data bus (such as the 8088 or 80188). These processors always fetch data one byte at a time, regardless of the alignment. However, using **EVEN** can speed certain operation on processors that have a 16-bit data bus (such as the 8086, 80186, or 80286), since the processor can fetch a word if the data is word aligned, but must do two memory fetches if the data is not word aligned. Similarly, using **ALIGN 4** can speed some operations with a 80386 processor, since the processor can fetch four bytes at a time if the data is doubleword aligned.

Note

The **ALIGN** directive is a new feature of Version 5.0 of the Microsoft Macro Assembler. In previous versions, data could be word aligned by using the **EVEN** directive, but other alignments could not be specified.

The **EVEN** directive should not be used in segments with **BYTE** align type. Similarly, the *number* specified with the **ALIGN** directive should be at least equal to the size of the align type of the segment where the directive is given.

■ Example

```
                DOSSEG
                .MODEL  small
                .STACK  100h
                .DATA
                .
                .
                .
                ALIGN   4                       ; For faster data access
stuff           DW      66,124,573,99,75
                .
                .
                .
                ALIGN   4                       ; For faster data access
evenstuff       DW      ?,?,?,?,?
                .CODE
start:          mov     ax,@data                ; Load segment location
                mov     ds,ax                   ;   into DS
                mov     es,ax                   ;   and ES registers

                mov     cx,5                    ; Load count
                mov     si,OFFSET stuff         ; Point to source
                mov     di,OFFSET evenstuff;      and destination
                ALIGN   4                       ; Align for faster loop access
mloop:          lodsw                           ; Load a word
                inc     ax                      ; Make it even by incrementing
                and     ax,NOT 1                ;   and turning off first bit
                stosw                           ; Store
                loop    mloop                   ; Again
```

In this example, the words at `stuff` and `evenstuff` are forced to doubleword boundaries. This makes access to the data faster with processors that have either a 32-bit or 16-bit data bus. Without this alignment, the initial data might start on an odd boundary and the processor would have to fetch half of each word at a time with a 16-bit data bus or half of each doubleword with a 32-bit data bus.

Similarly, the alignment in the code segment speeds up repeated access to the code at the start of the loop. The sample code sacrifices program size in order to achieve significant speed improvements on the 80386 and more moderate improvements on the 8086 and 80286. There is no speed advantage on the 8088.

CHAPTER 7

USING STRUCTURES AND RECORDS

7.1 Structures 143
- 7.1.1 Declaring Structure Types 143
- 7.1.2 Defining Structure Variables 145
- 7.1.3 Using Structure Operands 146

7.2 Records 147
- 7.2.1 Declaring Record Types 148
- 7.2.2 Defining Record Variables 150
- 7.2.3 Using Record Operands and Record Variables 151
- 7.2.4 Record Operators 153
  - 7.2.4.1 The MASK Operator 153
  - 7.2.4.2 The WIDTH Operator 153
- 7.2.5 Using Record-Field Operands 154

The Macro Assembler can define and use two kinds of multifield variables: structures and records.

Structures are templates for data objects made up of smaller data objects. A structure can be used to define structure variables, which are made up of smaller variables called fields. Fields within a structure can be different sizes, and each can be accessed individually.

Records are templates for data objects whose bits can be described as groups of bits called fields. A record can be used to define record variables. Each bit field in a record variable can be used separately in constant operands or expressions. The processor cannot access bits individually at run time, but bit fields can be used with logical bit instructions to change bits indirectly.

This chapter describes structures and records and tells how to use them.

7.1 Structures

A structure variable is a collection of data objects that can be accessed symbolically as a single data object. Objects within the structure can have different sizes and can be accessed symbolically.

There are two steps in using structure variables:

1. Declare a structure type. A structure type is a template for data. It declares the sizes and, optionally, the initial values for objects in the structure. By itself the structure type does not define any data. The structure type is used by **MASM** during assembly but is not saved as part of the object file.
2. Define one or more variables having the structure type. For each variable defined, memory is allocated to the object file in the format declared by the structure type.

The structure variable can then be used as an operand in assembler statements. The structure variable can be accessed as a whole by using the structure name, or individual fields can be accessed by using structure and field names.

7.1.1 Declaring Structure Types

The **STRUC** and **ENDS** directives mark the beginning and end of a type declaration for a structure.

■ Syntax

name **STRUC**
fielddeclarations
name **ENDS**

The *name* declares the name of the structure type. It must be unique. The *fielddeclarations* declare the fields of the structure. Any number of field declarations may be given. They must follow the form of data definitions described in Section 6.3, "Defining and Initializing Data." Default initial values may be declared individually or with the **DUP** operator.

The names given to fields must be unique within the source file where they are declared. When variables are defined, the field names will represent the offset from the beginning of the structure to the corresponding field.

When declaring strings in a structure type, make sure the initial values are long enough to accommodate the largest possible string. Strings smaller than the field size can be placed in the structure variable, but larger strings will be truncated.

A structure declaration can contain field declarations and comments. Starting with Version 5.0 of the Macro Assembler, conditional-assembly statements are allowed in structure declarations. No other kinds of statements are allowed. Since the **STRUC** directive is not allowed inside structure declarations, structures cannot be nested.

Note

The **ENDS** directive that marks the end of a structure has the same mnemonic as the **ENDS** directive that marks the end of a segment. The assembler recognizes the meaning of the directive from context. Make sure each **SEGMENT** directive and each **STRUC** directive has its own **ENDS** directive.

■ Example

```
student     STRUC                ; Structure for student records
id          DW      ?            ; Field for identification #
sname       DB      "Last, First Middle    "
scores      DB      10 DUP (100)  ; Field for 10 scores
student     ENDS
```

Within the sample structure `student`, the fields `id`, `sname`, and `scores` have the offset values 0, 2, and 24, respectively.

7.1.2 Defining Structure Variables

A structure variable is a variable with one or more fields of different sizes. The sizes and initial values of the fields are determined by the structure type with which the variable is defined.

■ Syntax

[[*name*]] *structurename* <[[*initialvalue* [[,*initialvalue*...]]]>

The *name* is the name assigned to the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *structurename* is the name of a structure type previously declared by using the **STRUC** and **ENDS** directives.

An *initialvalue* can be given for each field in the structure. Its type must not be incompatible with the type of the corresponding field. The angle brackets (< >) are required even if no initial value is given. If *initialvalues* are given for more than one field, the values must be separated by commas.

If the **DUP** operator (see Section 6.3.2, "Arrays and Buffers") is used to initialize multiple structure variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of structure variables as shown below:

```
war          date     365 DUP (<,,1940>)
```

You need not initialize all fields in a structure. If an initial value is left blank, the assembler automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is undefined.

■ Examples

The following examples use the `student` type declared in the first example in Section 7.1.1, "Declaring Structure Types":

```
s1          student <>              ; Uses default values of type


s2          student <1467,"White, Robert D.",>
                                    ; Override default values of first two
                                    ;   fields--use default value of third

sarray      student 100 DUP (<>)    ; Declare 100 student variables
                                    ;   with default initial values
```

Note

You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was declared. For example, assume the following structure declaration:

```
stuff       STRUC
buffer      DB      100 DUP (?)       ; Can't override
crlf        DB      13,10             ; Can't override
query       DB      'Filename:        ; String <= can override
endmark     DB      36                ; Can override
stuff       ENDS
```

The `buffer` and `crlf` fields cannot be overridden by initial values in the structure definition because they have multiple values. The `query` field can be overridden as long as the overriding string is no longer than `query` (10 bytes). A longer string would generate an error. The `endmark` field can be overridden by any byte value.

7.1.3 Using Structure Operands

Like other variables, structure variables can be accessed by name. Fields within structure variables can also be accessed by using the syntax shown below:

■ Syntax

variable.field

The *variable* must be the name of a structure (or an operand that resolves to the address of a structure). The *field* must be the name of a field within that structure. The *variable* is separated from *field* by a period. The period is discussed as a structure field-name operator in Section 9.2.1.2.

The address of a structure operand is the sum of the offsets of *variable* and *field*. The address is relative to the segment or group in which the variable is declared.

■ Examples

```
date        STRUC                       ; Declare structure
month       DB      ?
day         DB      ?
year        DW      ?
date        ENDS

            .DATA
yesterday   date    <9,30,1987>         ; Declare structure
today       date    <10,1,1987>         ;   variables
tomorrow    date    <10,2,1987>

            .CODE
            .
            .
            .
            mov     al,yesterday.day    ; Use structure variables
            mov     ah,today.month      ;   as operands
            mov     tomorrow.year,dx
            mov     bx,OFFSET yesterday ; Load structure address
            mov     ax,[bx].month       ; Use as indirect operand
            .
            .
            .
```

7.2 Records

A record variable is a byte or word variable in which specific bit fields can be accessed symbolically. Records can be doubleword variables with the 80386 processor. Bit fields within the record can have different sizes.

There are two steps in declaring record variables:

1. Declare a record type. A record type is a template for data. It declares the sizes and, optionally, the initial values for bit fields in the record. By itself the record type does not define any data. The record type is used by **MASM** during assembly but is not saved as part of the object file.
2. Define one or more variables having the record type. For each variable defined, memory is allocated to the object file in the format declared by the type.

The record variable can then be used as an operand in assembler statements. The record variable can be accessed as a whole by using the record name, or individual fields can be specified by using the record name and a field name combined with the field-name operator. A record type can also be used as a constant (immediate data).

7.2.1 Declaring Record Types

The **RECORD** directive declares a record type for an 8- or 16-bit record that contains one or more bit fields. With the 80386, 32-bit records can also be declared.

■ Syntax

recordname **RECORD** *field* [[,*field*...]]

The *recordname* is the name of the record type to be used when creating the record. The *field* declares the name, width, and initial value for the field.

The syntax for each *field* is shown below:

■ Syntax

fieldname:*width*[[= *expression*]]

The *fieldname* is the name of a field in the record, *width* is the number of bits in the field, and *expression* is the initial (or default) value for the field.

Any number of *field* combinations can be given for a record, as long as each is separated from its predecessor by a comma. The sum of the widths for all fields must not exceed 16 bits.

The width must be a constant. If the total width of all declared fields is larger than eight bits, then the assembler uses two bytes. Otherwise, only one byte is used.

80386 Only

Records can be up to 32 bits in width when the 80386 processor is enabled with **.386**. If the total width is 8 bits or less, the assembler uses 1 byte; if the width is 9 to 16 bytes, the assembler uses 2 bytes; and if the width is larger than 16 bits, the assembler uses 4 bytes.

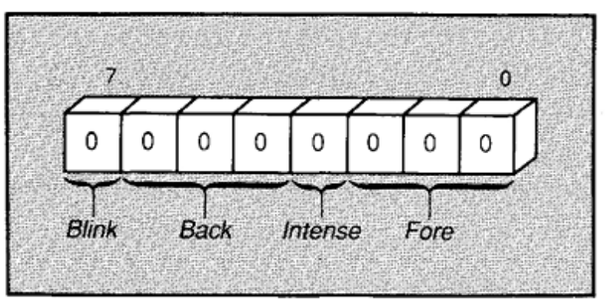
If *expression* is given, it declares the initial value for the field. An error message is generated if an initial value is too large for the width of its field. If the field is at least seven bits wide, you can use an ASCII character for *expression*. The expression must not contain a forward reference to any symbol.

In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record are initialized to 0.

■ Example 1

```
color       RECORD  blink:1,back:3,intense:1,fore:3
```

The example above creates a byte record type `color` having four fields: `blink`, `back`, `intense`, and `fore`. The contents of the record type are shown below:

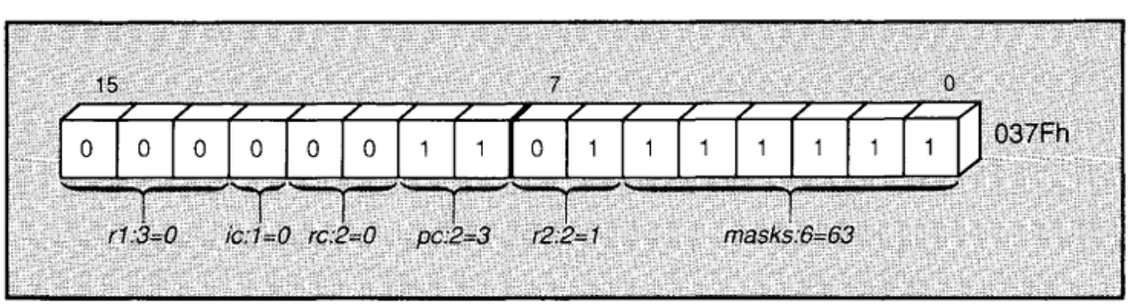


Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data are created.

■ Example 2

```
cw          RECORD   r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
```

Example 2 creates a record type `cw` having six fields. Each record declared by using this type occupies 16 bits of memory. The bit diagram below shows the contents of the record type:



Default values are given for each field. They can be used when data is declared for the record.

7.2.2 Defining Record Variables

A record variable is an 8-bit or 16-bit variable whose bits are divided into one or more fields. With the 80386, 32-bit variables are also allowed.

■ Syntax

[[*name*]] *recordname* <[[*initialvalue* [[,*initialvalue*...]]]>

The *name* is the symbolic name of the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *recordname* is the name of a record type that was previously declared by using the **RECORD** directive.

An *initialvalue* for each field in the record can be given as an integer, character constant, or an expression that resolves to a value compatible with the size of the field. Angle brackets (< >) are required even if no initial value is given. If initial values for more than one field are given, the values must be separated by commas.

If the **DUP** operator (see Section 6.3.2, "Arrays and Buffers") is used to initialize multiple record variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of record variables as shown below:

```
xmas        color   50 DUP (<1,2,0,4>)
```

You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is declared by the record type. If there is no default value, each bit in the field is cleared.

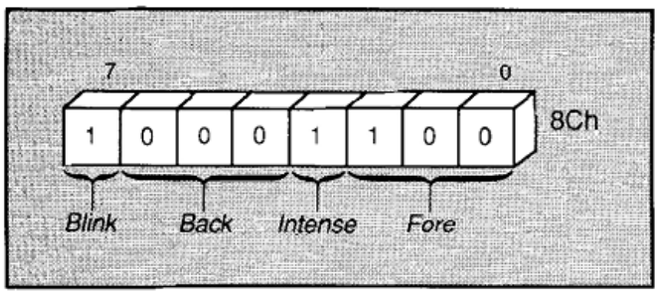
Sections 7.2.3, "Using Record Operands and Record Variables," and 7.2.4, "Record Operators," illustrate ways to use record data after it has been declared.

■ Examples

```
color       RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
warning     color   <1,0,1,4>                       ; Record definition
```

The definition above creates a variable named `warning` whose type is given by the record type `color`. The initial values of the fields in the variable are set to the values given in the record definition. The initial values

would override the default record values, had any been given in the declaration. The contents of the record variable are shown below:



■ Example 2

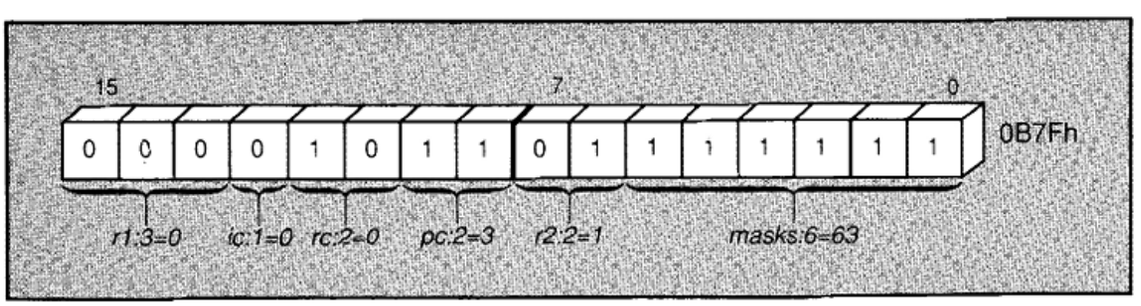
```
color       RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
colors      color   16 DUP (<>)                      ; Record declaration
```

Example 2 creates an array named `colors` containing 16 variables of type `color`. Since no initial values are given in either the declaration or the definition, the variables have undefined (0) values.

■ Example 3

```
cw          RECORD   r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
newcw       cw       <,,2,,,>
```

Example 3 creates a variable named `newcw` with type `cw`. The default values set in the type declaration are used for all fields except the `pc` field. This field is set to 2. The contents of the variable are shown below:



7.2.3 Using Record Operands and Record Variables

A record operand refers to the value of a record type. It should not be confused with a record variable. A record operand is a constant; a record variable is a value stored in memory. A record operand can be used with the following syntax:

■ Syntax

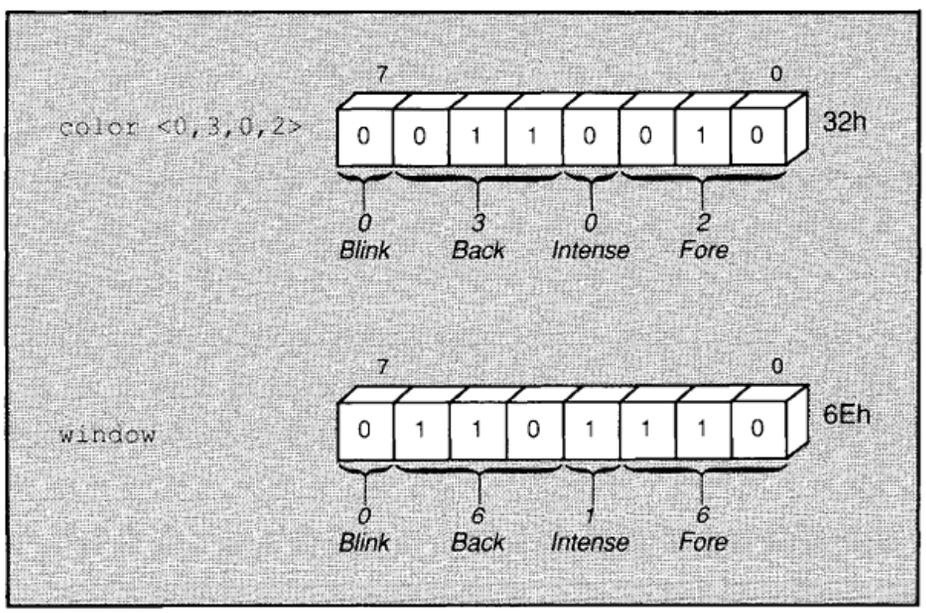
recordname <[[*value*][,*value...*]]>

The *recordname* must be the name of a record type declared in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, each value must be separated by a comma. Values can include expressions or symbols that evaluate to constants. The enclosing angle brackets (< >) are required, even if no value is given. If no value for a field is given, the default value for that field is used.

■ Example

```
            .DATA
color       RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
window      color   <0,6,1,6>                       ; Record definition

            .CODE
            .
            .
            .
            mov     ah,color <0,3,0,2> ; Load record operand
                                       ;  (constant value 32h)
            mov     bh,window          ; Load record variable
                                       ;  (memory value 6Eh)
```

In this example, the record operand `color <0,3,0,2>` and the record variable `warning` are loaded into registers. The contents of the values are shown below:



7.2.4 Record Operators

The **WIDTH** and **MASK** operators are used exclusively with records to return constant values representing different aspects of previously declared records.

7.2.4.1 The MASK Operator

The **MASK** operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a field bit. All other bits contain 0.

■ Syntax

MASK {*recordfieldname* | *record*}

The *recordfieldname* may be the name of any field in a previously defined record. The *record* may be the name of any previously defined record. The **NOT** operator is sometimes used with the **MASK** operator to reverse the bits of a mask.

■ Example

```
            .DATA
color       RECORD  blink:1,back:3,intense:1,fore:3
message     color   <0,5,1,1>
            .CODE
            .
            .
            .
            mov     ah,message          ; Load initial    0101 1001
            and     ah,NOT MASK back    ; Turn off    AND 1000 1111
                                        ; "back"          ---------
                                        ;                 0000 1001
            or      ah,MASK blink       ; Turn on      OR 1000 0000
                                        ; "blink"         ---------
                                        ;                 1000 1001
            xor     ah,MASK intense     ; Toggle      XOR 0000 1000
                                        ; "intense"       ---------
                                        ;                 1000 0001
```

7.2.4.2 The WIDTH Operator

The **WIDTH** operator returns the width (in bits) of a record or record field.

■ Syntax

WIDTH {*recordfieldname* | *record*}

The *recordfieldname* may be the name of any field defined in any record. The *record* may be the name of any defined record.

Note that the width of a field is the number or bits assigned for that field; the value of the field is the starting position (from the right) of the field.

■ Examples

```
            .DATA
color       RECORD  blink:1,back:3,intense:1,fore:3

wblink      EQU     WIDTH blink     ; "wblink"    = 1   "blink"    = 7
wback       EQU     WIDTH back      ; "wback"     = 3   "back"     = 4
wintense    EQU     WIDTH intense   ; "wintense"  = 1   "intense"  = 3
wfore       EQU     WIDTH fore      ; "wfore"     = 3   "fore"     = 0
wcolor      EQU     WIDTH color     ; "wcolor"    = 8

prompt      color   <1,5,1,1>

            .CODE
            .
            .
            .
            IF      (WIDTH color) GE 8 ; If color is 16 bit, load
            mov     ax,prompt          ;   into 16-bit register
            ELSE                       ; else
            mov     al,prompt          ;   load into low 8-bit register
            xor     ah,ah              ;   and clear high 8-bit register
            ENDIF
```

7.2.5 Using Record-Field Operands

Record-field operands represent the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand. The field name must be from a previously declared record.

Record-field operands are often used with the **WIDTH** and **MASK** operators, as described in Sections 7.2.4.1 and 7.2.4.2.

■ Example

```
        .DATA
color   RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
cursor  color   <1,5,1,1>                      ; Record definition
        .CODE
        .
        .
```

```
; Rotate "back" of "cursor" without changing other values

        mov     al,cursor           ; Load value from memory
        mov     ah,al               ; Save a copy for work       1101 1001=ah/al
        and     al,NOT MASK back    ; Mask out old bits     and  1000 1111=mask
                                    ;   to save old cursor       ---------
                                    ;                            1000 1001=al
        mov     cl,back             ; Load bit position
        shr     ah,cl               ; Shift to right             0000 1101=ah
        inc     ah                  ; Increment                  0000 1110=ah

        shl     ah,cl               ; Shift left again           1110 0000=ah
        and     ah,MASK back        ; Mask off extra bits   and  0111 0000=mask
                                    ;   to get new cursor        ---------
                                    ;                            0110 0000 ah
        or      ah,al               ; Combine old and new    or  1000 1001 al
                                    ;                            ---------
        mov     cursor,ah           ; Write back to memory       1110 1001 ah
```

This example illustrates several ways in which record fields can be used as operands and in expressions.

CHAPTER 8

CREATING PROGRAMS FROM MULTIPLE MODULES

8.1 Declaring Symbols Public 160
8.2 Declaring Symbols External 161
8.3 Using Multiple Modules 164
8.4 Declaring Symbols Communal 165
8.5 Specifying Library Files 169

Most medium and large assembly-language programs are created from several source files or modules. When several modules are used, the scope of symbols becomes important. This chapter discusses the scope of symbols and explains how to declare global symbols that can be accessed from any module. It also tells you how to specify a module that will be accessed from a library.

Symbols such as labels and variable names can be either local or global in scope. By default, all symbols are local; they are specific to the source file in which they are defined. Symbols must be declared global if they must be accessed from modules other than the one in which they are defined.

To declare symbols global, they must be declared public in the source module in which they are defined. They must also be declared external in any module that must access the symbol. If the symbol represents uninitialized data, it can be declared communal—meaning that the symbol is both public and external. The **PUBLIC**, **EXTRN**, and **COMM** directives are used to declare symbols public, external, and communal, respectively.

Notes

The term "local" has a different meaning in assembly language than in many high-level languages. Often, local symbols in compiled languages are symbols that are known only within a procedure (called a function, routine, subprogram, or subroutine, depending on the language). Local symbols of this type cannot be declared by **MASM**, although procedures can be written to allocate local symbols dynamically at run time, as described in Section 17.4.4, "Using Local Variables."

By default, the assembler converts all lowercase letters in names declared with the **PUBLIC**, **EXTRN**, and **COMM** directives to uppercase letters before copying the name to the object file. The **/ML** and **/MX** options can be used in the **MASM** command line to direct the assembler to preserve lowercase letters when copying public and external symbols to the object file. This should be done when preparing assembler modules to be linked with modules from case-sensitive languages such as C.

8.1 Declaring Symbols Public

The **PUBLIC** directive is used to declare symbols public so that they can be accessed from other modules. If a symbol is not declared public, the symbol name is not written to the object file. The symbol has the value of its offset address during assembly, but the name and address are not available to the linker.

If the symbol is declared public, its name is associated with its offset address in the object file. During linking, symbols in different modules—but with the same name—are resolved to a single address.

Public symbol names are also used by some symbolic debuggers (such as **SYMDEB**) to associate addresses with symbols. However, variables and labels do not need to be declared public in order to be visible in the CodeView debugger.

■ Syntax

PUBLIC *name* [[,*name*]]...

The *name* must be the name of a variable, label, or numeric equate defined within the current source file. **PUBLIC** declarations can be placed anywhere in the source file. Equate names, if given, can only represent 1- or 2-byte integer or string values. Text macros (or text equates) cannot be declared public.

Note

Although absolute symbols can be declared public, aliases for public symbols should be avoided, since they may decrease the efficiency of the linker. For example, the following statements would increase processing time for the linker:

```
            PUBLIC  lines       ; Declare absolute symbol public
lines       EQU     rows        ; Declare alias for lines
rows        EQU     25          ; Assign value to alias
```

■ **Example**

```
            PUBLIC  true,status,first,clear
            .MODEL  small
true        EQU     -1
            .DATA
status      DB      1
            .CODE
            .
            .
            .
first       LABEL   FAR
clear       PROC
            .
            .
            .
clear       ENDP
```

8.2 Declaring Symbols External

If a symbol undeclared in a module must be accessed by instructions in that module, it must be declared with the **EXTRN** directive.

This directive tells the assembler not to generate an error, even though the symbol is not in the current module. The assembler assumes that the symbol occurs in another module. However, the symbol must actually exist and must be declared public in some module. Otherwise, the linker generates an error.

■ **Syntax**

EXTRN *name:type* [[,*name:type*]]...

The **EXTRN** directive defines an external variable, label, or symbol of the specified *name* and *type*. The *type* must match the type given to the item in its actual definition in some other module. It can be any one of the following:

Description	Types
Distance specifier	**NEAR**, **FAR**, or **PROC**
Size specifier	**BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**
Absolute	**ABS**

The **ABS** type is for symbols that represent constant numbers, such as equates declared with the **EQU** and = directives (see Section 11.1, "Using Equates").

The **PROC** type represents the default type for a procedure. For programs that use simplified segment directives, the type of an external symbol declared with **PROC** will be near for small or compact model, or far for medium, large, or huge model. Section 5.1.3, "Defining the Memory Model," tells you how to declare the memory model using the **.MODEL** directive. If full segment definitions are used, the default type represented by **PROC** is always near.

Although the actual address of an external symbol is not determined until link time, the assembler assumes a default segment for the item, based on where the **EXTRN** directive is placed in the source code. Placement of **EXTRN** directives should follow these rules.

- **NEAR** code labels (such as procedures) must be declared in the code segment from which they are accessed.
- **FAR** code labels can be declared anywhere in the source code. It may be convenient to declare them in the code segment from which they are accessed if the label may be **FAR** in one context or **NEAR** in another.
- Data must be declared in the segment in which it occurs. This may require that you define a dummy data segment for the external declaration.
- Absolute symbols can be declared anywhere in the source code.

■ Example 1

```
                EXTRN   max:ABS,act:FAR     ; Constant or FAR label anywhere
                DOSSEG
                .MODEL  small
                .STACK  100h
                .DATA
                EXTRN   nvar:BYTE           ; NEAR variable in near data
                .FARDATA
                EXTRN   fvar:WORD           ; FAR variable in far data

                .CODE
                EXTRN   task:PROC           ; PROC or NEAR in near code
start:          mov     ax,@data            ; Load segment
                mov     ds,ax               ;   into DS
                ASSUME  es:@fardata         ; Tell assembler
                mov     ax,@fardata         ; Tell processor that ES
                mov     es,ax               ;   has far data segment
                .
                .
                .
```

```
            mov     ah,nvar             ; Load external NEAR variable
            mov     bx,fvar             ; Load external FAR variable
            mov     cx,max              ; Load external constant
            call    task                ; Call procedure (NEAR or FAR)
            jmp     act                 ; Jump to FAR label

            END     start
```

Example 1 shows how each type of external symbol could be declared and used in a small-model program that uses simplified segment directives. Notice the use of the **PROC** type specifier to make the external-procedure memory model independent. The jump and its external declaration are written so that they will be **FAR** regardless of the memory model. Using these techniques, you can change the memory model without breaking code.

■ Example 2

```
            EXTRN   max:ABS,act:FAR     ; Constant or FAR label anywhere
            DOSSEG
STACK       SEGMENT PARA STACK 'STACK'
            DB      100h DUP (?)
STACK       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
            EXTRN   nvar:BYTE           ; NEAR variable in near data
_DATA       ENDS
FAR_DATA    SEGMENT PARA 'FAR_DATA'
            EXTRN   fvar:WORD           ; FAR variable in far data
FAR_DATA    ENDS

DGROUP      GROUP   _DATA,STACK
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            EXTRN   task:NEAR           ; NEAR procedure in near code
            ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
start:      mov     ax,DGROUP           ; Load segment
            mov     ds,ax               ;   into DS
            ASSUME  es:FAR_DATA         ; Tell assembler
            mov     ax,FAR_DATA         ; Tell processor that ES
            mov     es,ax               ;   has far data segment
            .
            .
            .
            mov     ah,nvar             ; Load external NEAR variable
            mov     bx,fvar             ; Load external FAR variable
            mov     cx,max              ; Load external constant

            call    task                ; Call NEAR procedure

            jmp     act                 ; Jump to FAR label

_TEXT       ENDS
            END     start
```

Example 2 shows a fragment similar to the one in Example 2, but with full segment definitions. Notice that the types of code labels must be declared specifically. If you wanted to change the memory model, you would have to specifically change each external declaration and each call or jump.

8.3 Using Multiple Modules

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules called `hello` and `display`.

The `hello` module is the program's initializing module. Execution starts at the instruction labeled `start` in the `hello` module. After initializing the data segment, the program calls the procedure `display` in the `display` module, where a DOS call is used to display a message on the screen. Execution then returns to the address after the call in the `hello` module.

The `hello` module is shown below:

```
            TITLE   hello
            DOSSEG
            .MODEL  small
            .STACK  256

            .DATA
            PUBLIC  message, lmessage
message     DB      "Hello, world.",13,10
lmessage    EQU     $ - message

            .CODE
            EXTRN   display:PROC        ; Declare in near code segment
start:      mov     ax,@data            ; Load segment location
            mov     ds,ax               ;   into DS register

            call    display             ; Call other module

            mov     ax,04C00h           ; Terminate with exit code 0
            int     21h                 ; Call DOS

            END     start               ; Start address in main module
```

The `display` module is shown below:

```
            TITLE   display
            EXTRN   lmessage:ABS        ; Declare anywhere
            .MODEL  small
            .DATA
            EXTRN   message:BYTE        ; Declare in near data segment

            .CODE
            PUBLIC  display
display     PROC
            mov     bx,1                ; File handle for standard output
            mov     cx,lmessage         ; Message length
            mov     dx,OFFSET message   ; Message address
            mov     ah,40h              ; Write function
            int     21h                 ; Call DOS
            ret
display     ENDP
            END                         ; No start address in second module
```

The sample program is a variation of the `hello.asm` program used in examples in Chapter 1, "Getting Started," except that it uses an external procedure to display to the screen. Notice that all symbols defined in one module but used in another are declared **PUBLIC** in the defining module and declared **EXTRN** in the using module.

For instance, `message` and `lmessage` are declared **PUBLIC** in `hello` and declared **EXTRN** in `display`. The procedure `display` is declared **EXTRN** in `hello` and **PUBLIC** in `display`.

To create an executable file for these modules, assemble each module separately, as in the following command lines:

```
MASM hello;
MASM display;
```

Then link the two modules:

```
LINK hello display;
```

The result is the executable file `hello.exe`.

For each source module, **MASM** writes a module name to the object file. The module name is used by some debuggers and by the linker when it displays error messages. Starting with Version 5.0, the module name is always the base name of the source module file. With previous versions, the module name could be specified with the **NAME** or **TITLE** directive.

For compatibility, **MASM** recognizes the **NAME** directive. However, **NAME** has no effect. Arguments to the directive are ignored.

8.4 Declaring Symbols Communal

Communal variables are uninitialized variables that are both public and external. They are often declared in include files.

If a variable must be used by several assembly routines, you can declare the variable communal in an include file, and then include the file in each of the assembly routines. Although the variable is declared in each source module, it exists at only one address. Using a communal variable in an include file and including it in several source modules is an alternative to defining the variable and declaring it public in one source module and then declaring it external in other modules.

If a variable is declared communal in one module and public in another, the public declaration takes precedence and the communal declaration has the same effect as an external declaration.

■ Syntax

COMM *definition*[**,***definition*]...

Each *definition* has the following syntax:

[**NEAR** | **FAR**] *label*:*size*[:*count*]

A communal variable can be **NEAR** or **FAR**. If neither is specified, the type will be that of the default memory model. If you use simplified segment directives, the default type is **NEAR** for small and medium models, or **FAR** for compact, large, and huge models. If you use full segment definitions the default type is **NEAR**.

The *label* is the name of the variable. The *size* can be **BYTE**, **WORD**, **DWORD**, **QWORD**, or **TBYTE**. The *count* is the number of elements. If no *count* is given, one element is assumed. Multiple variables can be defined with one **COMM** statement by separating each variable with a comma.

Note

C variables declared outside functions (except static variables) are communal unless explicitly initialized; they are the same as assembly-language communal variables. If you are writing assembly-language modules for C, you can declare the same communal variables in C include files and in **MASM** include files.

MASM cannot tell whether a communal variable has been used in another module. Allocation of communal variables is handled by **LINK**. As a result, communal variables have the following limitations that other variables declared in assembly language do not have:

- Communal variables cannot be initialized. Under DOS, initial values are not guaranteed to be 0 or any other value. The variables can be used for data, such as file buffers, that are not given a value until run time.

- Communal variables are not guaranteed to be allocated in the sequence in which they are declared. Assembly-language techniques that depend on the sequence and position in which data is defined

should not be used with communal variables. For example, the following statements do not work:

```
          COMM    buffer:WORD:128
lbuffer   EQU     $ - buffer ; "lbuffer" won't have desired value

bbuffer   LABEL   BYTE       ; "bbuffer" won't have desired address
          COMM    wbuffer:WORD:128
```

- Placement of communal declarations follows the same rules as external declarations. They must be declared inside a data segment. Examples of near and far communal variables are shown below:

```
.DATA
COMM    NEAR nbuffer:BYTE:30
.FARDATA
COMM    FAR fbuffer:WORD:40
```

- Communal variables are allocated in segments that are part of the Microsoft segment conventions. You cannot override the default to place communal variables in other segments.

  Near communal variables are placed in a segment called **c_common**, which is part of **DGROUP**. This group is created and initialized automatically if you use simplified segment directives. If you use full segment directives, you must create a group called **DGROUP** and use the **ASSUME** directive to associate it with the **DS** register.

  Far communal variables are placed in a segment called **FAR_BSS**. This segment has combine type private and class type **'FAR_BSS'**. This means that multiple segments with the same name can be created. Such segments cannot be accessed by name. They must be initialized indirectly using the **SEG** operator. For example, if a far communal variable (with word size) is called `fcomvar`, its segment can be initialized with the following lines:

```
ASSUME  ds:SEG comvar       ; Tell the assembler
mov     ax,SEG comvar       ; Tell the processor
mov     ds,ax
mov     bx,comvar           ; Use the variable
```

■ Example 1

```
IF      @datasize
.FARDATA
ELSE
.DATA
ENDIF
COMM    var:WORD, buffer:BYTE:10
```

Example 1 creates two communal variables. The first is a word variable called `var`. The second is a 10-byte array called `buffer`. Both have the

default size associated with the memory model of the program in which they are used.

■ Example 2

```
            .DATA
            COMM    temp:BYTE:128

ASCIIZ      MACRO   address          ; Name of address for string
            mov     temp,128         ; Insert maximum length
            mov     dx,OFFSET temp   ; Address of string buffer
            mov     ah,0Ah           ; Get string
            int     21h              ;
            mov     dl,temp[1]       ; Get length of string
            xor     dh,dh
            mov     bx,dx
            mov     temp[bx+2],0     ; Overwrite CR with null
address     EQU     OFFSET temp+2
            ENDM
```

Example 2 shows an include file that declares a buffer for temporary data. The buffer is then used in a macro in the same include file. An example of how the macro could be used in a source file is shown below:

```
            DOSSEG
            .MODEL  small
            INCLUDE communal.inc
            .DATA
message     DB      "Enter file name: $"
            .CODE
            .
            .
            .
            mov     dx,OFFSET message  ; Load offset of file prompt
            mov     ah,09h             ; Display prompt
            int     21h

            ASCIIZ  place              ; Get file name and
                                       ;   return address as "place"

            mov     al,00000010b       ; Load access code
            mov     dx,place           ; Load address of ASCIIZ string
            mov     ah,3Dh             ; Open the file
            int     21h
```

Note that once the macro is written, the user does not need to know the name of the temporary buffer or how it is used in the macro.

8.5 Specifying Library Files

The **INCLUDELIB** directive instructs the linker to link with a specified library file. If you are writing a program that calls library routines, you can use this directive to specify the library file in the assembly source file rather than in the **LINK** command line.

■ Syntax

INCLUDELIB *libraryname*

The *libraryname* is written to the comment record of the object file. The Intel title for this record is **COMENT**. At link time, the linker reads this record and links with the specified library file.

The *libraryname* must be a file name rather than a complete file specification. If you do not specify an extension, the default extension **.LIB** is assumed. **LINK** searches for the library file in the following order:

1. In the current directory
2. In any directories given in the library field of the **LINK** command line
3. In any directories listed in the **LIB** environment variable

■ Example

```
INCLUDELIB graphics
```

This statement passes a message from **MASM** telling **LINK** to use library routines from the file `graphics.lib`. If this statement is included in a source file called `draw`, then the program might be linked with the following command line:

```
LINK draw;
```

Without the **INCLUDELIB** directive, the program would have to be linked with the following command line:

```
LINK draw,,,graphics;
```

CHAPTER 9

USING OPERANDS AND EXPRESSIONS

9.1 Using Operands with Directives 173

9.2 Using Operators 174
- 9.2.1 Calculation Operators 174
  - 9.2.1.1 Arithmetic Operators 175
  - 9.2.1.2 Structure-Field-Name Operator 176
  - 9.2.1.3 Index Operator 177
  - 9.2.1.4 Shift Operators 178
  - 9.2.1.5 Bitwise Logical Operators 179
- 9.2.2 Relational Operators 180
- 9.2.3 Segment-Override Operator 181
- 9.2.4 Type Operators 182
  - 9.2.4.1 PTR Operator 182
  - 9.2.4.2 SHORT Operator 183
  - 9.2.4.3 THIS Operator 183
  - 9.2.4.4 HIGH and LOW Operators 184
  - 9.2.4.5 SEG Operator 184
  - 9.2.4.6 OFFSET Operator 185
  - 9.2.4.7 .TYPE Operator 186
  - 9.2.4.8 TYPE Operator 187
  - 9.2.4.9 LENGTH Operator 188
  - 9.2.4.10 SIZE Operator 188
- 9.2.5 Operator Precedence 189

9.3 Using the Location Counter 190

9.4 Using Forward References 191
- 9.4.1 Forward References to Labels 192
- 9.4.2 Forward References to Variables 194

9.5 Strong Typing for Memory Operands 194

Operands are the arguments that define values to be acted on by instructions or directives. Operands can be constants, variables, expressions, or keywords, depending on the instruction or directive, and the context of the statement.

A common type of operand is an expression. An expression consists of several operands that are combined to describe a value or memory location. Operators indicate the operations to be performed when combining the operands of an expression.

Expressions are evaluated at assembly time. By using expressions, you can instruct the assembler to calculate values that would be difficult or inconvenient to calculate when you are writing source code.

This chapter discusses operands, expressions, and operators as they are evaluated at assembly time. See Chapter 14, "Using Addressing Modes," for a discussion of the addressing modes that can be used to calculate operand values at run time. This chapter also discusses the location-counter operand, forward references, and strong typing of operands.

9.1 Using Operands with Directives

Each directive requires a specific type of operand. Most directives take string or numeric constants, or symbols or expressions that evaluate to such constants.

The type of operand varies for each directive, but the operand must always evaluate to a value that is known at assembly time. This differs from instructions, whose operands may not be known at assembly time and may vary at run time. Operands used with instructions are discussed in Chapter 14, "Using Addressing Modes."

Some directives, such as those used in data declarations, accept labels or variables as operands. When a symbol that refers to a memory location is used as an operand to a directive, the symbol represents the address of the symbol rather than its contents. This is because the contents may change at run time and are therefore not known at assembly time.

■ Example 1

```
        ORG     100h            ; Set address to 100h
var     DB      10h             ; Address of "var" is 100h
                                ; Value of "var" is 10h
pvar    DW      var             ; Address of "pvar" is 101h
                                ; Value of "pvar" is
                                ;   address of "var" (100h)
```

In Example 1, the operand of the **DW** directive in the third statement represents the address of `var` (100h) rather than its contents (10h). The address is relative to the start of the segment in which `var` is defined.

■ **Example 2**

```
            TITLE   doit                    ; String
_TEXT       SEGMENT BYTE PUBLIC 'CODE'      ; Key words
            INCLUDE \include\bios.inc       ; Pathname
            .RADIX  16                      ; Numeric constant
tst         DW      a / b                   ; Numeric expression
            PAGE    +                       ; Special character
sum         EQU     x * y                   ; Numeric expression
here        LABEL   WORD                    ; Type specifier
```

Example 2 illustrates the different kinds of values that can be used as directive operands.

9.2 Using Operators

The assembler provides a variety of operators for combining, comparing, changing, or analyzing operands. Some operators work with integer constants, some with memory values, and some with both. Operators cannot be used with floating-point constants since **MASM** does not recognize real numbers in expressions.

It is important to understand the difference between operators and instructions. Operators handle calculations of constant values that are known at assembly time. Instructions handle calculations of values that may not be known until run time. For example, the addition operator (+) handles assembly-time addition, while the **ADD** and **ADC** instructions handle run-time addition.

This section describes the different kinds of operators used in assembly-language statements and gives examples of expressions formed with them. In addition to the operators described in this chapter, you can use the **DUP** operator (Section 6.3.2, "Arrays and Buffers") the record operators (Section 7.2.5, "Using Record-Field Operands"), and the macro operators (Section 11.4, "Using Macro Operators").

9.2.1 Calculation Operators

MASM provides the common arithmetic operators as well as several other operators for adding, shifting, or doing bit manipulations. The sections below describe operators that can be used for doing numeric calculations.

Note

Constant values used with calculation operators are extended to 33 bits before the calculations are done. This rule applies regardless of the processor used. Exceptions are noted to this rule.

9.2.1.1 Arithmetic Operators

MASM recognizes a variety of arithmetic operators for common mathematical operations. Table 9.1 lists the arithmetic operators.

Table 9.1
Arithmetic Operators

Operator	Syntax	Meaning
+	*+expression*	Positive (unary)
–	*–expression*	Negative (unary)
*	*expression1***expression2*	Multiplication
/	*expression1*/*expression2*	Integer division
MOD	*expression1*MOD*expression2*	Remainder (modulus)
+	*expression1*+*expression2*	Addition
–	*expression1*–*expression2*	Subtraction

For all arithmetic operators except the addition operator (+) and the subtraction operator (–), the expressions operated on must be integer constants.

The addition and subtraction operators can be used to add or subtract an integer constant and a memory operand. The result can be used as a memory operand.

The subtraction operator can also be used to subtract one memory operand from another, but only if the operands refer to locations within the same segment. The result will be a constant, not a memory operand.

Note

The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to

designate addition or subtraction). The unary plus and minus have a higher level of precedence, as described in Section 9.2.5, "Operator Precedence."

■ Example 1

```
intgr       =       14  *  3            ; = 42
intgr       =       intgr /  4          ; 42 / 4 = 10
intgr       =       intgr  MOD  4       ; 10 mod 4 = 2
intgr       =       intgr +  4          ; 2 + 4 = 6
intgr       =       intgr -  3          ; 6 - 3 = 3
intgr       =       -intgr - 8          ; -3 - 8 = -11
intgr       =       -intgr - intgr      ; -11 - 11 = 22
```

Example 1 illustrates arithmetic operators used in integer expressions.

■ Example 2

```
            ORG     100h
a           DB      ?                   ; Address is 100h
b           DB      ?                   ; Address is 101h
mem1        EQU     a + 5               ; mem1 = 100h + 5 = 105h
mem2        EQU     a - 5               ; mem2 = 100h - 5 = OFBh
const       EQU     b - a               ; const = 101h - 100h = 1
```

Example 2 illustrates arithmetic operators used in memory expressions.

9.2.1.2 Structure-Field-Name Operator

The structure-field-name operator (.) indicates addition. It is used to designate a field within a structure.

■ Syntax

variable.field

The *variable* is a memory operand (usually a previously declared structure variable) and *field* is the name of a field within the structure. See Section 7.1, "Structures," for more information.

■ Example

```
            .DATA
date        STRUC                       ; Declare structure
month       DB      ?
day         DB      ?
year        DW      ?
date        ENDS
yesterday   date    <12,31,1987>        ; Define structure variables
today       date    <1,1,1988>

            .CODE
            .
            .
            .
            mov     bh,yesterday.day    ; Load structure variable

            mov     bx,OFFSET today     ; Load structure variable address
            inc     [bx].year           ; Use in indirect memory operand
```

9.2.1.3 Index Operator

The index operator ([ ]) indicates addition. It is similar to the addition (+) operator.

■ Syntax

[[*expression1*]][*expression2*]

In most cases *expression1* is simply added to *expression2*. The limitations of the addition operator for adding memory operands also apply to the index operator. For example, two direct memory operands cannot be added. The expression `label1[label2]` is illegal if both are memory operands.

The index operator has an extended function in specifying indirect memory operands. Section 14.3.2, "Indirect Memory Operands," explains the use of indirect memory operands. The index brackets must be outside the register or registers that specify the indirect displacement. However, any of the three operators that indicate addition (the addition operator, the index operator, or the structure-field-name operator) may be used for multiple additions within the expression.

For example, the following statements are equivalent:

```
mov     ax,table[bx][di]
mov     ax,table[bx+di]
mov     ax,[table+bx+di]
mov     ax,[table][bx][di]
```

The following statements are illegal because the index operator does not enclose the registers that specify indirect displacement:

```
mov     ax,table+bx+di      ; Illegal - no index operator
mov     ax,[table]+bx+di    ; Illegal - registers not
                            ;  inside index operator
```

The index operator is typically used to index elements of a data object, such as variables in an array or characters in a string.

■ Example 1

```
mov     al,string[3]        ; Get 4th element of string
add     ax,array[4]         ; Add 5th element of array
mov     string[7],al        ; Load into 8th element of string
```

Example 1 illustrates the index operator used with direct memory operands.

■ Example 2

```
mov     ax,[bx]             ; Get element BX points to
add     ax,array[si]        ; Add element SI points to
mov     string[di],al       ; Load element DI points to
cmp     cx,table[bx][di]    ; Compare to element BX and DI
                            ;  point to
```

Example 2 illustrates the index operator used with indirect memory operands.

9.2.1.4 Shift Operators

The **SHR** and **SHL** operators can be used to shift bits in constant values. Both perform logical shifts. Bits on the right for **SHL** and on the left for **SHR** are zero-filled as their contents are shifted out of position.

■ Syntax

expression **SHR** *count*
expression **SHL** *count*

The *expression* is shifted right or left by *count* number of bits. Bits shifted off either end of the expression are lost. If *count* is greater than or equal to 16 (32 on the 80386), the result is 0.

Do not confuse the **SHR** and **SHL** operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time. The assembler can tell the difference between instructions and operands from context.

■ **Examples**

```
mov     ax,01110111b SHL 3 ; Load 011101110000b
mov     ah,01110111b SHR 3 ; Load 01110b
```

9.2.1.5 Bitwise Logical Operators

The bitwise operators perform logical operations on each bit of an expression. The expressions must resolve to constant values. Table 9.2 lists the logical operators and their meanings.

Table 9.2
Logical Operators

Operator	Syntax	Meaning
NOT	**NOT** *expression*	Bitwise complement
AND	*expression1* **AND** *expression2*	Bitwise AND
OR	*expression1* **OR** *expression2*	Bitwise inclusive OR
XOR	*expression1* **XOR** *expression2*	Bitwise exclusive OR

Do not confuse the **NOT**, **AND**, **OR**, and **XOR** operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time. The assembler can tell the difference between instructions and operands from context.

Note

Although calculations on expressions using the **AND**, **OR**, and **XOR** operators are done using 33-bit numbers, the results are truncated to 32 bits. Calculations on expressions using the **NOT** operator are truncated to 16 bits (except on the 80386).

■ **Examples**

```
mov     ax,NOT 11110000b              ; Load 1111111100001111b
mov     ah,NOT 11110000b              ; Load 00001111b
mov     ah,01010101b AND 11110000b    ; Load 01010000b
mov     ah,01010101b OR  11110000b    ; Load 11110101b
mov     ah,01010101b XOR 11110000b    ; Load 10100101b
```

9.2.2 Relational Operators

The relational operators compare two expressions and return true (–1) if the condition specified by the operator is satisfied, or false (0) if it is not. The expressions must resolve to constant values. Relational operators are typically used with conditional directives. Table 9.3 lists the operators and the values they return if the specified condition is satisfied.

Table 9.3
Relational Operators

Operator	Syntax	Returned Value
EQ	*expression1* **EQ** *expression2*	True if expressions are equal
NE	*expression1* **NE** *expression2*	True if expressions are not equal
LT	*expression1* **LT** *expression2*	True if left expression is less than right
LE	*expression1* **LE** *expression2*	True if left expression is less than or equal to right
GT	*expression1* **GT** *expression2*	True if left expression is greater than right
GE	*expression1* **GE** *expression2*	True if left expression is greater than or equal to right

Note

The **EQ** and **NE** operators treat their arguments as 32-bit numbers. Numbers specified with the 32nd bit set are considered negative. For example, the expression `-1 EQ 0FFFFFFFFh` is true, but the expression `-1 NE 0FFFFFFFFh` is false.

The **LT**,**LE**,**GT**, and **GE** operators treat their arguments as 33-bit numbers, in which the 33rd bit specifies the sign. For example, `0FFFFFFFFh` is 4,294,967,295, not -1. The expression `1 GT -1` is true, but the expression `1 GT 0FFFFFFFFh` is false.

■ **Examples**

```
mov     ax,4 EQ 3  ; Load false ( 0)
mov     ax,4 NE 3  ; Load true  (-1)
mov     ax,4 LT 3  ; Load false ( 0)
mov     ax,4 LE 3  ; Load false ( 0)
mov     ax,4 GT 3  ; Load true  (-1)
mov     ax,4 GE 3  ; Load true  (-1)
```

9.2.3 Segment-Override Operator

The segment-override operator (:) forces the address of a variable or label to be computed relative to a specific segment.

■ **Syntax**

segment:expression

The *segment* can be specified in several ways. It can be one of the segment registers: **CS**, **DS**, **SS**, or **ES** (or **FS** or **GS** on the 80386). It can also be a segment or group name. In this case, the name must have been previously defined with a **SEGMENT** or **GROUP** directive and assigned to a segment register with an **ASSUME** directive. The expression can be a constant, expression, or a **SEG** expression. See Section 9.2.4.5 for more information on the **SEG** operator.

Note

When a segment override is given with an indexed operand, the segment must be specified outside the index operators. For example, `es:[di]` is correct, but `[es:di]` generates an error.

■ **Examples**

```
mov     ax,ss:[bx+4]        ; Override default assume (DS)
mov     al,es:082h          ; Load from ES

ASSUME  ds:FAR_DATA         ; Tell the assembler and
mov     bx,FAR_DATA:count   ;   load from a far segment
```

As shown in the last two statements, a segment override with a segment name is not enough if no segment register is assumed for the segment name. You must use the **ASSUME** statement to assign a segment register, as explained in Section 5.4, "Associating Segments with Registers."

9.2.4 Type Operators

This section describes the assembler operators that specify or analyze the types of memory operands and other expressions.

9.2.4.1 PTR Operator

The **PTR** operator specifies the type for a variable or label.

■ Syntax

type **PTR** *expression*

The operator forces *expression* to be treated as having *type*. The *expression* can be any operand. The *type* can be **BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE** for memory operands. It can be **NEAR**, **FAR**, or **PROC** for labels.

The **PTR** operator is typically used with forward references to define explicitly what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. See Section 9.4 for more information on forward references.

The **PTR** operator is also used to enable instructions to access variables in ways that would otherwise generate errors. For example, you could use the **PTR** operator to access the high-order byte of a **WORD** size variable. The **PTR** operator is required for **FAR** calls and jumps to forward-referenced labels.

■ Example 1

```
            .DATA
stuff       DD      ?
buffer      DB      20 DUP (?)

            .CODE
            .
            .
            .
            call    FAR PTR task            ; Call a far procedure
            jmp     FAR PTR place           ; Jump far

            mov     bx,WORD PTR stuff[0]    ; Load a word from a
                                            ;   doubleword variable
            add     ax,WORD PTR buffer[bx]  ; Add a word from a
                                            ;   byte variable
```

9.2.4.2 SHORT Operator

The **SHORT** operator sets the type of a specified label to **SHORT**. Short labels can be used in **JMP** instructions whenever the distance from the label to the instruction is less than 128 bytes.

■ Syntax

SHORT *label*

Instructions using short labels are a byte smaller than identical instructions using the default near labels. See Section 9.4.1, "Forward Reference to Labels," for information on using the **SHORT** operator with jump instructions.

■ Example

```
            jmp     again               ; Jump 128 bytes or more
            .
            .
            .
            jmp     SHORT again         ; Jump less than 128 bytes
            .
            .
            .
again:
```

9.2.4.3 THIS Operator

The **THIS** operator creates an operand whose offset and segment values are equal to the current location-counter value and whose type is specified by the operator.

■ Syntax

THIS *type*

The *type* can be **BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE** for memory operands. It can be **NEAR**, **FAR**, or **PROC** for labels.

The **THIS** operator is typically used with the **EQU** or equal-sign (=) directive to create labels and variables. The result is similar to using the **LABEL** directive.

■ Examples

```
tag1        EQU     THIS BYTE   ; Both represent the same variable
tag2        LABEL   BYTE

check1      EQU     THIS NEAR   ; All represent the same address
check2      LABEL   NEAR
check3:
check4      PROC    NEAR
check4      ENDP
```

9.2.4.4 HIGH and LOW Operators

The **HIGH** and **LOW** operators return the high and low bytes, respectively, of an expression.

■ Syntax

HIGH *expression*
LOW *expression*

The **HIGH** operator returns the high-order eight bits of *expression*; the **LOW** operator returns the low-order eight bits. The *expression* must evaluate to a constant. You cannot use the **HIGH** and **LOW** operators on the contents of a memory operand since the contents may change at run time.

■ Examples

```
stuff       EQU     OABCDh
            mov     ah,HIGH stuff       ; Load OABh
            mov     al,LOW stuff        ; Load OCDh
```

9.2.4.5 SEG Operator

The **SEG** operator returns the segment address of an expression.

■ Syntax

SEG *expression*

The *expression* can be any label, variable, segment name, group name, or other memory operand. The **SEG** operator cannot be used with constant expressions. The returned value can be used as a memory operand.

■ Examples

```
                .DATA
var             DB      ?
                .CODE
                .
                .
                .
                mov     ax,SEG var          ; Get address of segment
                                            ;   where variable is declared
                ASSUME  ds:SEG var          ; Assume segment of variable
```

9.2.4.6 OFFSET Operator

The **OFFSET** operator returns the offset address of an expression.

■ Syntax

OFFSET *expression*

The *expression* can be any label, variable, or other direct memory operand. Constant expressions return meaningless values. The value returned by the **OFFSET** operand is an immediate (constant) operand.

If simplified segment directives are given, the returned value varies. If the item is declared in a near data segment, the returned value is the number of bytes between the item and the beginning of its group (normally **DGROUP**). If the item is declared in a far segment, the returned value is the number of bytes between the item and the beginning of the segment.

If full segment definitions are given, the returned value is a memory operand equal to the number of bytes between the item and the beginning of the segment in which it is defined.

The segment-override operator (:) can be used to force **OFFSET** to return the number of bytes between the item in *expression* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group when full segment definitions are used. For example, the statement

```
mov     bx,OFFSET DGROUP:array
```

is not the same as

```
mov     bx,OFFSET array
```

if `array` is not the first segment in DGROUP.

■ Examples

```
            .DATA
string      DB      "This is it."
            .CODE
            .
            .
            .
            mov     dx,OFFSET string    ; Load offset of variable
```

9.2.4.7 .TYPE Operator

The **.TYPE** operator returns a byte that defines the mode and scope of an expression.

■ Syntax

.TYPE *expression*

If the *expression* is not valid, **.TYPE** returns 0. Otherwise **.TYPE** returns a byte having the bit setting shown in Table 9.4. Only bits 0, 1, 5, and 7 are affected. Other bits are always 0.

Table 9.4
.TYPE Operator and Variable Attributes

Bit Position	If Bit = 0	If Bit = 1
0	Not program related	Program related
1	Not data related	Data related
5	Not defined	Defined
7	Local or public scope	External scope

The **.TYPE** operator is typically used in macros in which different kinds of arguments may need to be handled differently.

■ Example

```
display     MACRO   string
            IF      ((.TYPE string) SHL 14) NE 8000h
            IF2
            %OUT    Argument must be a variable
            ENDIF
            ENDIF
            mov     dx,OFFSET string
            mov     ah,09h
            int     21h
            ENDM
```

This macro checks to see if the argument passed to it is data related (a variable). It does this by shifting all bits except the relevant bits (1 and 0) left so that they can be checked. If the data bit is not set, an error message is generated.

9.2.4.8 TYPE Operator

The **TYPE** operator returns a number that represents the type of an expression.

■ Syntax

TYPE *expression*

If *expression* evaluates to a variable, the operator returns the number of bytes in each data object in the variable. Each byte in a string is considered a separate data object, so the **TYPE** operator returns 1 for strings.

If *expression* evaluates to a structure or structure variable, the operator returns the number of bytes in the structure. If *expression* is a label, the operator returns 0FFFFh for **NEAR** labels and 0FFFEh for **FAR** labels. If *expression* is a constant, the operator returns 0.

The returned value can be used to specify the type for a **PTR** operator.

■ Examples

```
            .DATA
var         DW      ?
array       DD      10 DUP (?)
str         DB      "This is a test"
            .CODE
            .
            .
            .
            mov     ax,TYPE var             ; Puts 2 in AX
            mov     bx,TYPE array           ; Puts 4 in BX
            mov     cx,TYPE str             ; Puts 1 in CX
            jmp     (TYPE room) PTR room    ; Jump is near or far,
            .                               ;   depending on memory model
            .
            .
room        LABEL   PROC
```

9.2.4.9 LENGTH Operator

The **LENGTH** operator returns the number of data elements in an array or other variable defined with the **DUP** operator.

■ Syntax

LENGTH *variable*

The returned value is the number of elements of the declared size in the variable. If the variable was declared with nested **DUP** operators, only the value given for the outer **DUP** operator is returned. If the variable was not declared with the **DUP** operator, the value returned is always 1.

■ Examples

```
array       DD      100 DUP(0FFFFFFh)
table       DW      100 DUP(1,10 DUP(?))
string      DB      'This is a string'
var         DT      ?
larray      EQU     LENGTH array        ; 100 - number of elements
ltable      EQU     LENGTH table        ; 100 - inner DUP not counted
lstring     EQU     LENGTH string       ; 1 - string is one element
lvar        EQU     LENGTH var          ; 1
            .
            .
            .
            mov     cx,LENGTH array     ; Load number of elements
again:      .                           ; Perform some operation on
            .                           ;   each element
            .
            loop    again
```

9.2.4.10 SIZE Operator

The **SIZE** operator returns the total number of bytes allocated for an array or other variable defined with the **DUP** operator.

■ Syntax

SIZE *variable*

The returned value is equal to the value of **LENGTH** *variable* times the value of **TYPE** *variable*. If the variable was declared with nested **DUP** operators, only the value given for the outside **DUP** operator is considered. If the variable was not declared with the **DUP** operator, the value returned is always **TYPE** *variable*.

■ Example

```
array       DD      100 DUP(1)
table       DW      100 DUP(1,10 DUP(?))
string      DB      'This is a string'
var         DT      ?
sarray      EQU     SIZE array          ; 400 - elements times size
stable      EQU     SIZE table          ; 200 - inner DUP ignored
sstring     EQU     SIZE string         ; 1 - string is one element
svar        EQU     SIZE var            ; 10 - bytes in variable
            .
            .
            .
            mov     cx,SIZE array       ; Load number of bytes
again:      .                           ; Perform some operation on
            .                           ;   each byte
            .
            loop    again
```

9.2.5 Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.
- The order of evaluation can be overridden by using parentheses. Operations in parentheses are always performed before any adjacent operations.

The order of precedence for all operators is listed in Table 9.5. Operators on the same line have equal precedence.

Table 9.5

Operator Precedence

Precedence	Operators
(Highest)	
1	**LENGTH, SIZE, WIDTH, MASK,** (), [], <>
2	. (structure-field-name operator)
3	:
4	**PTR, OFFSET, SEG, TYPE, THIS**
5	**HIGH, LOW**
6	+,– (unary)
7	*,/, **MOD, SHL, SHR**
8	+, – (binary)
9	**EQ, NE, LT, LE, GT, GE**
10	**NOT**
11	**AND**
12	**OR, XOR**
13	**SHORT, .TYPE**
(Lowest)	

■ Examples

```
a       EQU     8 / 4 * 2           ; Equals 4
b       EQU     8 / (4 * 2)         ; Equals 1
c       EQU     8 + 4 * 2           ; Equals 16
d       EQU     (8 + 4) * 2         ; Equals 24
e       EQU     8 OR 4 AND 2        ; Equals 8
f       EQU     (8 OR 4) AND 3      ; Equals 0
```

9.3 Using the Location Counter

The location counter is a special operand that, during assembly, represents the address of the statement currently being assembled. At assembly time, the location counter keeps changing, but when used in source code it resolves to a constant representing an address.

The location counter has the same attributes as a near label. It represents an offset that is relative to the current segment and is equal to the number of bytes generated for the segment to that point.

■ Example 1

```
string      DB      "Who wants to count every byte in a string, "
            DB      "especially if you might change it later."
lstring     EQU     $-string    ; Let the assembler do it
```

Example 1 shows one way of using the location-counter operand in expressions relating to data.

■ Example 2

```
            cmp     ax,bx
            jl      shortjump   ; If ax < bx, go to "shortjump"
            .                   ;   else if ax >= bx, continue
            .
shortjump:  .

            cmp     ax,bx
            jge     $+5         ; If ax >= bx, continue
            jmp     longjump    ;   else if ax < bx, go to "longjump"
            .                   ; This is "$+5"
            .
longjump:   .
```

Example 2 illustrates how you can use the location counter to do conditional jumps of more than 128 bytes. The first part shows the normal way of coding jumps of less than 128 bytes, and the second part shows how to code the same jump when the label is more than 128 bytes away.

9.4 Using Forward References

The assembler permits you to refer to labels, variable names, segment names, and other symbols before they are declared in the source code. Such references are called forward references.

The assembler handles forward references by making assumptions about them on the first pass and then attempting to correct the assumptions, if necessary, on the second pass. Checking and correcting assumptions on the second pass takes processing time, so source code with forward references assembles more slowly than source code with no forward references.

In addition, the assembler may make incorrect assumptions that it cannot correct, or corrects at a cost in program efficiency.

9.4.1 Forward References to Labels

Forward references to labels may result in incorrect or inefficient code.

In the statement below, the label `target` is a forward reference:

```
            jmp     target              ; Generates 3 bytes
            .                           ;   in 16-bit segment
            .
            .
target:
```

Since the assembler processes source files sequentially, `target` is unknown when it is first encountered. Assuming 16-bit segments, it could be one of three types: short (–128 to 127 bytes from the jump), near (–32,768 to 32,767 bytes from the jump), or far (in a different segment than the jump). **MASM** assumes that `target` is a near label, and assembles the number of bytes necessary to specify a near label: one byte for the instruction and two bytes for the operand.

If on the second pass the assembler learns that `target` is a short label, it will need only two bytes: one for the instruction and one for the operand. However, it will not be able to change its previous assembly and the three-byte version of the assembly will stand. If the assembler learns that `target` is a far label, it will need five bytes. Since it can't make this adjustment, it will generate a phase error.

You can override the assembler's assumptions by specifying the exact size of the jump. For example, if you know that a **JMP** instruction refers to a label less than 128 bytes from the jump, you can use the **SHORT** operator, as shown below:

```
            jmp     SHORT target        ; Generates 2 bytes
            .                           ;   in 16-bit segment
            .
            .
target:
```

Using the **SHORT** operator makes the code smaller and slightly faster. If the assembler has to use the three-byte form when the two-byte form would be acceptable, it will generate a warning message if the warning level is 2. (The warning level can be set with the **/W** option, as described in Section 2.4.13.) You can ignore the warning, or you can go back to the source code and change the code to eliminate the forward references.

Note

The **SHORT** operator in the example above would not be needed if `target` were located before the jump. The assembler would have already processed `target` and would be able to make adjustments based on its distance.

If you use the **SHORT** operator when the label being jumped to is more than 128 bytes away, **MASM** generates an error message. You can either remove the **SHORT** operator, or try to reorganize your program to reduce the distance.

If a far jump to a forward-referenced label is required, you must override the assembler's assumptions with the **FAR** and **PTR** operators, as shown below:

```
            jmp     FAR PTR target      ; Generates 5 bytes
            .                           ;   in 16-bit segment
            .
            .
target:                                 ; In different segment
```

If the type of a label has been established earlier in the source code with an **EXTRN** directive, the type does not need to be specified in the jump statement.

■ 80386 Only

If the 80386 processor is enabled, jumps with forward references have different limitations. One difference is that conditional jumps can be either short or near. With previous processors, all conditional jumps were short. For 32-bit segments, the number of bytes generated for near and far jumps is greater in order to handle the larger addresses in the operand.

■ Example 1

```
.MODEL  large               ; Model comes first, so use
.386                        ;   16-bit segments
.CODE
.
.
.
jmp     SHORT place         ; Short unconditional jump - 2 bytes
jne     SHORT place         ; Short conditional jump - 2 bytes
jmp     place               ; Near unconditional jump - 3 bytes
jne     place               ; Near conditional jump - 4 bytes
jmp     FAR PTR place       ; Far unconditional jump - 5 bytes
```

- **Example 2**

```
.386                            ; .386 comes first, so use
.MODEL  large                   ;   32-bit segments
.CODE
.
.
.
jmp     SHORT place             ; Short unconditional jump - 2 bytes
jne     SHORT place             ; Short conditional jump - 2 bytes
jmp     place                   ; Near unconditional jump - 5 bytes
jne     place                   ; Near conditional jump - 6 bytes
jmp     FAR PTR place           ; Far unconditional jump - 7 bytes
```

9.4.2 Forward References to Variables

When **MASM** encounters code referencing variables that have not yet been defined in Pass 1, it makes assumptions about the segment where the variable will be defined. If on Pass 2 the assumptions turn out to be wrong, an error will occur.

These problems usually occur with complex segment structures that do not follow the Microsoft segment conventions. The problems never appear if simplified segment directives are used.

By default, **MASM** assumes that variables are referenced to the **DS** register. If a statement must access a variable in a segment not associated with the **DS** register, and if the variable has not been defined earlier in the source code, you must use the segment-override operator to specify the segment.

The situation is different if neither the variable nor the segment in which it is defined has been defined earlier in the source code. In this case, you must assign the segment to a group earlier in the source code. **MASM** will then know about the existence of the segment even though it has not yet been defined.

9.5 Strong Typing for Memory Operands

The assembler carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that when an instruction uses two operands with implied data types, the operand types must match. Warning messages are generated for nonmatching types.

For example, in the following fragment, the variable `string` is incorrectly used in a move instruction:

```
            .DATA
string      DB      "A message."
            .CODE
            .
            .
            .
            mov     ax,string[1]
```

The AX register has **WORD** type, but `string` has **BYTE** type. Therefore, the statement generates warning message 37:

```
Operand types must match
```

To avoid all ambiguity and prevent the warning error, use the **PTR** operator to override the variable's type, as shown below:

```
            mov     ax,WORD PTR string[1]
```

You can ignore the warnings if you are willing to trust the assembler's assumptions. When a register and memory operand are mixed, the assembler assumes that the register operand is always the correct size. For example, in the statement

```
            mov     ax,string[1]
```

the assembler assumes that the programmer wishes the word size of the register to override the byte size of the variable. A word starting at `string[1]` will be moved into **AX**. In the statement

```
            mov     string[1],ax
```

the assembler assumes that the programmer wishes to move the word value in **AX** into the word starting at `string[1]`. However, the assembler's assumptions are not always as clear as in these examples. You should not ignore warnings about type mismatches unless you are sure you understand how your code will be assembled.

Note

Some assemblers (including early versions of the IBM Macro Assembler) do not do strict type checking. For compatibility with these assemblers, type errors are warnings rather than severe errors. Many assembly-language program listings in books and magazines are written for assemblers with weak type checking. Such programs may produce warning messages, but assemble correctly. You can use the **/W** option to turn off type warnings if you are sure the code is correct.

CHAPTER 10

ASSEMBLING CONDITIONALLY

10.1 Using Conditional-Assembly Directives.....199
- 10.1.1 Testing Expressions with IF and IFE Directives 200
- 10.1.2 Testing the Pass with IF1 and IF2 Directives 201
- 10.1.3 Testing Symbol Definition with IFDEF and IFNDEF Directives.....201
- 10.1.4 Verifying Macro Parameters with IFB and IFNB Directives.....202
- 10.1.5 Comparing Macro Arguments with IFIDN and IFDIF Directives.....203

10.2 Using Conditional-Error Directives.....204
- 10.2.1 Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 205
- 10.2.2 Testing Expressions with .ERRE or .ERRNZ Directives 206
- 10.2.3 Verifying Symbol Definition with .ERRDEF and .ERRNDEF Directives..207
- 10.2.4 Testing for Macro Parameters with .ERRB and .ERRNB Directives.....207
- 10.2.5 Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives.....208

The Macro Assembler provides two types of conditional directives, conditional-assembly and conditional-error directives. Conditional-assembly directives test for a specified condition and assemble a block of statements if the condition is true. Conditional-error directives test for a specified condition and generate an assembly error if the condition is true.

Both kinds of conditional directives test assembly-time conditions. They cannot test run-time conditions. Only expressions that evaluate to constants during assembly can be compared or tested.

Since macros and conditional-assembly directives are often used together, you may need to refer to Chapter 11, "Using Equates, Macros, and Repeat Blocks," to understand some of the examples in this chapter. In particular, conditional directives are frequently used with the special macro operators described in Section 11.4, "Using Macro Operators."

10.1 Using Conditional-Assembly Directives

The conditional-assembly directives include the following:

IF	**IFDEF**	**IFNB**
IF1	**IFDIF**	**IFNDEF**
IF2	**IFE**	**ENDIF**
IFB	**IFIDN**	**ELSE**

The **IF** directives and the **ENDIF** and **ELSE** directives can be used to enclose the statements to be considered for conditional assembly.

■ Syntax

IF *condition*
statements
[**ELSE**
statements]
ENDIF

The *statements* following the **IF** directive can be any valid statements, including other conditional blocks. The **ELSE** directive and its *statements* are optional. **ENDIF** ends the block.

The statements in the conditional block are assembled only if the condition specified by the corresponding **IF** statement is satisfied. If the conditional block contains an **ELSE** directive, only the statements up to the **ELSE** directive are assembled. The statements that follow the **ELSE** directive are assembled only if the **IF** statement is not met. An **ENDIF** directive must mark the end of any conditional-assembly block. No more than one **ELSE** directive is allowed for each **IF** statement.

IF statements can be nested up to 255 levels. A nested **ELSE** directive always belongs to the nearest preceding **IF** statement that does not have its own **ELSE**.

10.1.1 Testing Expressions with IF and IFE Directives

The **IF** and **IFE** directives test the value of an expression and grant assembly based on the result.

■ Syntax

IF *expression*
IFE *expression*

The **IF** directive grants assembly if the value of *expression* is true (nonzero). The **IFE** directive grants assembly if the value of *expression* is false (0). The *expression* must resolve to a constant value and must not contain forward references.

■ Example

```
IF      debug GT 20
push    debug
call    adebug
ELSE
call    bdebug
ENDIF
```

In this example, a different debug routine will be called, depending on the value of debug.

10.1.2 Testing the Pass with IF1 and IF2 Directives

The **IF1** and **IF2** directives test the current assembly pass and grant assembly only on the pass specified by the directive. Multiple passes of the assembler are discussed in Section 2.5.7, "Reading a Pass 1 Listing."

■ Syntax

IF1
IF2

The **IF1** directive grants assembly only on Pass 1. **IF2** grants assembly only on Pass 2. The directives take no arguments.

Macros usually only need to be processed once. You can enclose blocks of macros in **IF1** blocks to prevent them from being reprocessed on the second pass.

■ Example

```
            IF1                 ; Define on first pass only
dostuff     MACRO   argument
            .
            .
            .
            ENDM
            ENDIF
```

10.1.3 Testing Symbol Definition with IFDEF and IFNDEF Directives

The **IFDEF** and **IFNDEF** directives test whether or not a symbol has been defined and grant assembly based on the result.

■ Syntax

IFDEF *name*
IFNDEF *name*

The **IFDEF** directive grants assembly only if *name* is a defined label, variable, or symbol. The **IFNDEF** directive grants assembly if *name* has not yet been defined.

The name can be any valid name. Note that if *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

■ Example

```
            IFDEF   buffer
buff        DB      buffer DUP(?)
            ENDIF
```

In this example, `buff` is allocated only if `buffer` has been previously defined.

One way to use this conditional block is to leave `buffer` undefined in the source file and define it if needed by using the **/D***symbol* option (see Section 2.4.4, "Defining Assembler Symbols.") when you start **MASM**. For example, if the conditional block is in `test.asm`, you could start the assembler with the following command line:

```
MASM /Dbuffer=1024 test;
```

The command line would define the symbol `buffer`; as a result, the conditional assemble would allocate `buff`. However, if you didn't need `buff`, you could use the following command line:

```
MASM test;
```

10.1.4 Verifying Macro Parameters with IFB and IFNB Directives

The **IFB** and **IFNB** directives test to see if a specified argument was passed to a macro and grant assembly based on the result.

■ Syntax

IFB <*argument*>
IFNB <*argument*>

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The **IFB** directive grants assembly if *argument* is blank. The **IFNB** directive grants assembly if *argument* is not blank. The arguments can be any name, number, or expression. Angle brackets (< >) are required.

■ **Example**

```
Write       MACRO   buffer,bytes,handle
            IFNB    <handle>
            mov     bx,handle       ; (1=stdout,2=stderr,3=aux,4=printer)
            ELSE
            mov     bx,1            ; Default standard out
            ENDIF
            mov     dx,OFFSET buffer; Address of buffer to write to
            mov     cx,bytes        ; Number of bytes to write
            mov     ah,40h
            int     21h
            ENDM
```

In this example, a default value is used if no value is specified for the third macro argument.

10.1.5 Comparing Macro Arguments with IFIDN and IFDIF Directives

The **IFIDN** and **IFDIF** directives compare two macro arguments and grant assembly based on the result.

■ **Syntax**

IFIDN[**I**] <*argument1*>,<*argument2*>
IFDIF[**I**] <*argument1*>,<*argument2*>

These directives are always used inside macros, and they always test whether real arguments passed for two specified arguments are the same. The **IFIDN** directive grants assembly if *argument1* and *argument2* are identical. The **IFDIF** directive grants assembly if *argument1* and *argument2* are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional **I** at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same will be evaluated the same, regardless of case. This is a new feature starting with Version 5.0. If the **I** is not given, the directive is case sensitive.

■ **Example**

```
divide8     MACRO   numerator,denominator
            IFDIFI  <numerator>,<al>    ;; If numerator isn't AL
            mov     al,numerator        ;;   make it AL
            ENDIF.
            xor     ah,ah
            div     denominator
            ENDM
```

In this example, a macro uses the **IFDIFI** directive to check one of the arguments and take a different action, depending on the text of the string. The sample macro could be enhanced further by checking for other values that would require adjustment (such as a denominator passed in **AL** or passed in **AH**).

10.2 Using Conditional-Error Directives

Conditional-error directives can be used to debug programs and check for assembly-time errors. By inserting a conditional-error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional-error directives to test for boundary conditions in macros.

The conditional-error directives and the error messages they produce are listed in Table 10.1.

Table 10.1
Conditional-Error Directives

Directive	Number	Message
.ERR1	87	Forced error - pass1
.ERR2	88	Forced error - pass2
.ERR	89	Forced error
.ERRE	90	Forced error - expression true (0)
.ERRNZ	91	Forced error - expression false (not 0)
.ERRNDEF	92	Forced error - symbol not defined
.ERRDEF	93	Forced error - symbol defined
.ERRB	94	Forced error - string blank
.ERRNB	95	Forced error - string not blank
.ERRIDN[I]	96	Forced error - strings identical
.ERRDIF[I]	97	Forced error - strings different

Like other severe errors, those generated by conditional-error directives cause the assembler to return exit code 7. If a severe error is encountered during assembly, **MASM** will delete the object module. All conditional error directives except **ERR1** generate severe errors.

10.2.1 Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 Directives

The **.ERR**, **.ERR1**, and **.ERR2** directives force an error where the directives occur in the source file. The error is generated unconditionally when the directive is encountered, but the directives can be placed within conditional-assembly blocks to limit the errors to certain situations.

■ Syntax

.ERR
.ERR1
.ERR2

The **.ERR** directive forces an error regardless of the pass. The **.ERR1** and **.ERR2** directives force the error only on their respective passes. The **.ERR1** directive appears only on the screen or in the listing file if you use the **/D** option to request a Pass 1 listing.

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

■ Example

```
IFDEF       dos
            .
            .
            .
ELSE
IFDEF       xenix
            .
            .
            .
            ELSE
            .ERR
            %OUT dos or xenix must be defined
            ENDIF
            ENDIF
```

This example makes sure that either the symbol `dos` or the symbol `xenix` is defined. If neither is defined, the nested **ELSE** condition is assembled and an error message is generated. Since the **.ERR** directive is used, an error would be generated on each pass. You could use **.ERR1** or **.ERR2** to check if you want the error to be generated only on the corresponding pass.

10.2.2 Testing Expressions with .ERRE or .ERRNZ Directives

The **.ERRE** and **.ERRNZ** directives test the value of an expression and conditionally generate an error based on the result.

■ Syntax

.ERRE *expression*
.ERRNZ *expression*

The **.ERRE** directive generates an error if the *expression* is false (0). The **.ERRNZ** directive generates an error if the *expression* is true (nonzero). The *expression* must resolve to a constant value and must not contain forward references.

■ Example

```
buffer      MACRO   count,bname
            .ERRE   count LE 128        ;; Allocate memory, but
bname       DB      count DUP(0)        ;;   no more than 128 bytes
            ENDM
            .
            .
            .
            buffer  128,buf1            ; Data allocated - no error
            buffer  129,buf2            ; Error generated
```

In this example, the **.ERRE** directive is used to check the boundaries of a parameter passed to the macro `buffer`. If `count` is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If `count` is greater than 128, the expression will be false (0) and the error will be generated.

10.2.3 Verifying Symbol Definition with .ERRDEF and .ERRNDEF Directives

The **.ERRDEF** and **.ERRNDEF** directives test whether or not a symbol is defined and conditionally generate an error based on the result.

■ Syntax

.ERRDEF *name*
.ERRNDEF *name*

The **.ERRDEF** directive produces an error if *name* is defined as a label, variable, or symbol. The **.ERRNDEF** directive produces an error if *name* has not yet been defined. If *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

■ Example

```
.ERRNDEF publevel
IF       publevel LE 2
PUBLIC   var1, var2
ELSE
PUBLIC   var1, var2, var3
ENDIF
```

In this example, the **.ERRNDEF** directive at the beginning of the conditional block makes sure that a symbol being tested in the block actually exists.

10.2.4 Testing for Macro Parameters with .ERRB and .ERRNB Directives

The **.ERRB** and **.ERRNB** directives test whether a specified argument was passed to a macro and conditionally generate an error based on the result.

■ Syntax

.ERRB <*argument*>
.ERRNB <*argument*>

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The

.ERRB directive generates an error if *argument* is blank. The **.ERRNB** directive generates an error if *argument* is not blank. The *argument* can be any name, number, or expression. Angle brackets (<>) are required.

■ Example

```
work        MACRO   realarg,testarg
            .ERRB   <realarg>  ;; Error if no parameters
            .ERRNB  <testarg>  ;; Error if more than one parameter
            .
            .
            .
            ENDM
```

In this example, error directives are used to make sure that one, and only one, argument is passed to the macro. The **.ERRB** directive generates an error if no argument is passed to the macro. The **.ERRNB** directive generates an error if more than one argument is passed to the macro.

10.2.5 Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives

The **.ERRIDN** and **.ERRDIF** directives compare two macro arguments and conditionally generate an error based on the result.

■ Syntax

.ERRIDN[**I**] <*argument1*>,<*argument2*>
.ERRDIF[**I**] <*argument1*>,<*argument2*>

These directives are always used inside macros, and they always compare the real arguments specified for two parameters. The **.ERRIDN** directive generates an error if the arguments are identical. The **.ERRDIF** directive generates an error if the arguments are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional **I** at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same will be evaluated

the same regardless of case. This is a new feature starting with Version 5.0. If the **I** is not given, the directive is case sensitive.

■ Example

```
addem       MACRO     ad1,ad2,sum
            .ERRIDNI <ax>,<ad2> ;; Error if ad2 is "ax"
            mov       ax,ad1    ;; Would overwrite if ad2 were AX
            add       ax,ad2
            mov       sum,ax    ;; Sum must be register or memory
            ENDM
```

In this example, the **.ERRIDNI** directive is used to protect against passing the **AX** register as the second parameter, since this would cause the macro to fail.

CHAPTER 11

USING EQUATES, MACROS, AND REPEAT BLOCKS

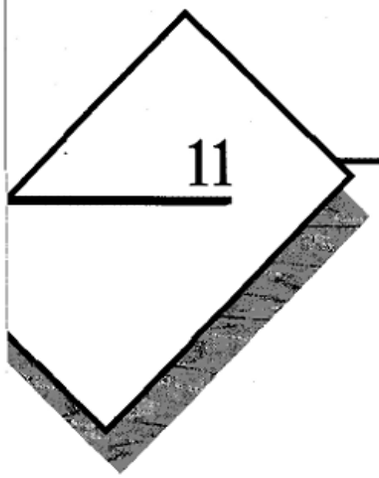
11.1 Using Equates 213
11.1.1 Redefinable Numeric Equates 213
11.1.2 Nonredefinable Numeric Equates 214
11.1.3 String Equates 216

11.2 Using Macros 217
11.2.1 Defining Macros 218
11.2.2 Calling Macros 219
11.2.3 Using Local Symbols 220
11.2.4 Exiting from a Macro 222

11.3 Defining Repeat Blocks 223
11.3.1 The REPT Directive 223
11.3.2 The IRP Directive 224
11.3.3 The IRPC Directive 225

11.4 Using Macro Operators 226
11.4.1 Substitute Operator 226
11.4.2 Literal-Text Operator 228
11.4.3 Literal-Character Operator 229
11.4.4 Expression Operator 230
11.4.5 Macro Comments 231

11.5 Using Recursive, Nested, and Redefined Macros 231
11.5.1 Using Recursion 231
11.5.2 Nesting Macro Definitions 232
11.5.3 Nesting Macro Calls 233
11.5.4 Redefining Macros 234
11.5.5 Avoiding Inadvertent Substitutions 234



CHAPTER 11

11.6 Managing Macros and Equates 235
11.6.1 Using Include Files.....235
11.6.2 Purging Macros from Memory.....237

This chapter explains how to use equates, macros, and repeat blocks. Equates are constant values assigned to symbols so that the symbol can be used in place of the value. Macros are a series of statements that are assigned a symbolic name (and optionally parameters) so that the symbol can be used in place of the statements. Repeat blocks are a special form of macro used to do repeated statements.

Both equates and macros are processed at assembly time. They can simplify writing source code by allowing the user to substitute mnemonic names for constants and repetitive code. By changing a macro or equate, a programmer can change the effect of statements throughout the source code.

In exchange for these conveniences, the programmer loses some assembly-time efficiency. Assembly may be slightly slower for a program that uses macros and equates extensively than for the same program written without them. However, the program without macros and equates usually takes longer to write and is more difficult to maintain.

11.1 Using Equates

The equate directives enable you to use symbols that represent numeric or string constants. **MASM** recognizes three kinds of equates:

1. Redefinable numeric equates
2. Nonredefinable numeric equates
3. String equates (also called text macros)

11.1.1 Redefinable Numeric Equates

Redefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol can be redefined at any point during assembly time. Although the value of a redefinable equate may be different at different points in the source code, a constant value will be assigned for each use, and that value will not change at run time.

Redefinable equates are often used for assembly-time calculations in macros and repeat blocks.

■ Syntax

name= expression

The equal-sign (=) directive creates or redefines a constant symbol by assigning the numeric value of *expression* to *name*. No storage is allocated for the symbol. The symbol can be used in subsequent statements as an immediate operand having the assigned value. It can be redefined at any time.

The *expression* can be an integer, a constant expression, a one- or two-character string constant (four-character on the 80386), or an expression that evaluates to an address. The *name* must be either a unique name or a name previously defined by using the equal-sign (=) directive.

Note

Redefinable equates must be assigned numeric values. String constants longer than two characters cannot be used.

■ Example

```
counter     =       0               ; Initialize counter
array       LABEL   BYTE            ; Label array of increasing numbers
            REPT    100             ; Repeat 100 times
            DB      counter         ; Initialize number
counter     =       counter + 1     ; Increment counter
            ENDM
```

This example redefines equates inside a repeat block to declare an array initialized to increasing values from 0 to 100. The equal-sign directive is used to increment the `counter` symbol for each loop. See Section 11.3 for more information on repeat blocks.

11.1.2 Nonredefinable Numeric Equates

Nonredefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol cannot be redefined.

Nonredefinable numeric equates are often used for assigning mnemonic names to constant values. This can make the code more readable and easier to maintain. If a constant value used in numerous places in the source code needs to be changed, then the equate can be changed in one place rather than throughout the source code.

■ Syntax

name **EQU** *expression*

The **EQU** directive creates constant symbols by assigning *expression* to *name*. The assembler replaces each subsequent occurrence of *name* with the value of *expression*. Once a numeric equate has been defined with the **EQU** directive, it cannot be redefined. Attempting to do so generates an error.

Note

String constants can also be defined with the **EQU** directive, but the syntax is different, as described in Section 11.1.3, "String Equates."

No storage is allocated for the symbol. Symbols defined with numeric values can be used in subsequent statements as immediate operands having the assigned value.

■ Examples

```
column      EQU     80                  ; Numeric constant 80
row         EQU     25                  ; Numeric constant 25
screenful   EQU     column * row        ; Numeric constant 2000
line        EQU     row                 ; Alias for "row"

            .DATA
buffer      DW      screenful

            .CODE
            .
            .
            .
            mov     cx,column
            mov     bx,line
```

11.1.3 String Equates

String equates (or text macros) are used to assign a string constant to a symbol. String equates can be used in a variety of contexts, including defining aliases and string constants.

■ **Syntax**

name **EQU** [[<]]*string*[[>]]

The **EQU** directive creates constant symbols by assigning *string* to *name*. The assembler replaces each subsequent occurrence of *name* with *string*. Symbols defined to represent strings with the **EQU** directive can be redefined to new strings. Symbols cannot be defined to represent strings with the equal-sign (=) directive.

An alias is a special kind of string equate. It is a symbol that is equated to another symbol or keyword.

Note

The use of angle brackets to force string evaluation is a new feature of Version 5.0 of the Macro Assembler. Previous versions tried to evaluate equates as expressions. If the string did not evaluate to a valid expression, **MASM** evaluated it as a string. This behavior sometimes caused unexpected consequences.

For example, the statement

```
rt          EQU     run-time
```

would be evaluated as `run` minus `time`, even though the user might intend to define the string `run-time`. If `run` and `time` were not already defined as numeric equates, the statement would generate an error. Using angle brackets solves this problem. The statement

```
rt          EQU     <run-time>
```

is evaluated as the string `run-time`.

When maintaining existing source code, you can leave string equates alone that evaluate correctly, but for new source code that will not be used with previous versions of **MASM**, it is a good idea to enclose all string equates in angle brackets.

■ Examples

```
; String equate definitions
pi          EQU     <3.1415>            ; String constant "3.1415"
prompt      EQU     <'Type Name: '>     ; String constant "'Type Name: '"
WPT         EQU     <WORD PTR>          ; String constant for "WORD PTR"
arg1        EQU     <[bp+4]>            ; String constant for "[bp+4]"

; Use of string equates
            .DATA
message     DB      prompt              ; Allocate string "Type Name: "
pie         DQ      pi                  ; Allocate real number 3.1415

            .CODE
            .
            .
            .
            inc     WPT parm1           ; Increment word value of
                                        ;   argument passed on stack
```

11.2 Using Macros

Macros enable you to assign a symbolic name to a block of source statements, and then to use that name in your source file to represent the statements. Parameters can also be defined to represent arguments passed to the macro.

Macro expansion is a text-processing function that occurs at assembly time. Each time **MASM** encounters the text associated with a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text of parameter names is replaced with the text of the corresponding actual arguments.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls the macro. Macros and equates are often kept in a separate file and made available to the program through an **INCLUDE** directive (see Section 11.6.1, "Using Include Files") at the start of the source code.

Note

Since most macros only need to be expanded once, you can increase efficiency by processing them only during a single pass of the assembler. You can do this by enclosing the macros (or an **INCLUDE** statement that calls them) in a conditional block using the **IF1** directive. Any macros that use the **EXTRN** or **PUBLIC** statements should be processed on Pass 1 rather than Pass 2 to increase linker efficiency.

Often a task can be done by using either a macro or procedure. For example, the addup procedure shown in Section 17.4.3, "Passing Arguments on the Stack," does the same thing as the addup macro in Section 11.2.1, "Defining Macros." Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures are coded only once in the executable file, but the increased overhead of saving and restoring addresses and parameters can make them slower.

The section below tells how to define and call macros. Repeat blocks, a special form of macro for doing repeated operations, are discussed separately in Section 11.3.

11.2.1 Defining Macros

The **MACRO** and **ENDM** directives are used to define macros. **MACRO** designates the beginning of the macro block and **ENDM** designates the end of the macro block.

■ Syntax

name **MACRO** [[*parameter* [[,*parameter*]]...]]
statements
ENDM

The *name* must be unique and a valid symbol name. It can be used later in the source file to invoke the macro.

The *parameters* (sometimes called dummy parameters) are names that act as placeholders for values to be passed as arguments to the macro when it is called. Any number of *parameters* can be specified, but they must all fit on one line. If you give more than one parameter, you must separate them with commas, spaces, or tabs. Commas can always be used as separators; spaces and tabs may cause ambiguity if the arguments are expressions.

Note

This manual uses the term "parameter" to refer to a placeholder for a value that will be passed to a macro or procedure. Parameters appear in macro or procedure definitions. The term "argument" is used to refer to an actual value passed to the macro or procedure when it is called.

Any valid assembler statement may be placed within a macro, including statements that call or define other macros. Any number of statements can

be used. The *parameters* can be used any number of times in the statements. Macros can be nested, redefined, or used recursively, as explained in Section 11.5, "Using Recursive, Nested, and Redefined Macros."

MASM assembles the statements in a macro only if the macro is called, and only at the point in the source file from which it is called. The macro definition itself is never assembled.

A macro definition can include the **LOCAL** directive, which lets you define labels used only within a macro, or the **EXITM** directive, which allows you to exit from a macro before all the statements in the block are expanded. These directives are discussed in Sections 11.2.3, "Using Local Symbols," and 11.2.4, "Exiting from a Macro." Macro operators can also be used in macro definitions, as described in Section 11.4, "Using Macro Operators."

■ Example

```
addup       MACRO   ad1,ad2,ad3
            mov     ax,ad1      ;; First parameter in AX
            add     ax,ad2      ;; Add next two parameters
            add     ax,ad3      ;;   and leave sum in AX
            ENDM
```

The preceding example defines a macro named addup, which uses three parameters to add three values and leave their sum in the **AX** register. The three parameters will be replaced with arguments when the macro is called.

11.2.2 Calling Macros

A macro call directs **MASM** to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

■ Syntax

name [[*argument* [[,*argument*]]...]]

The *name* must be the name of a macro defined earlier in the source file. The *arguments* can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs.

MASM replaces the first parameter with the first argument, the second parameter with the second argument, and so on. If a macro call has more

arguments than the macro has parameters, the extra arguments are ignored. If a call has fewer arguments than the macro has parameters, any remaining parameters are replaced with a null (empty) string.

You can use conditional statements to enable macros to check for null strings or other types of arguments. The macro can then take appropriate action to adjust to different kinds of arguments. See Chapter 10, "Assembling Conditionally," for more information on using conditional-assembly and conditional-error directives to test macro arguments.

■ Example

```
addup       MACRO   ad1,ad2,ad3         ; Macro definition
            mov     ax,ad1              ;; First parameter in AX
            add     ax,ad2              ;; Add next two parameters
            add     ax,ad3              ;;   and leave sum in AX
            ENDM
            .
            .
            .
            addup   bx,2,count          ; Macro call
```

When the addup macro is called, **MASM** replaces the parameters with the actual parameters given in the macro call. In the example above, the assembler would expand the macro call to the following code:

```
            mov     ax,bx
            add     ax,2
            add     ax,count
```

This code could be shown in an assembler listing, depending on whether the **.LALL**, **.XALL**, or **.SALL** directive was in effect (see Section 12.3.3, "Controlling Listing of Macros").

11.2.3 Using Local Symbols

The **LOCAL** directive can be used within a macro to define symbols that are available only within the defined macro.

Note

In this context, the term "local" is not related to the public availability of a symbol, as described in Chapter 8, "Creating Programs from Multiple Modules," or to variables that are defined to be local to a procedure, as described in Section 17.4.4, "Using Local Variables." "Local" simply means that the symbol is not known outside the macro where it is defined.

■ Syntax

LOCAL *localname* [[,*localname*]]...

The *localname* is a temporary symbol name that is to be replaced by a unique symbol name when the macro is expanded. At least one *localname* is required for each **LOCAL** directive. If more than one local symbol is given, the names must be separated with commas. Once declared, *localname* can be used in any statement within the macro definition.

MASM creates a new actual name for *localname* each time the macro is expanded. The actual name has the following form:

??number

The *number* is a hexadecimal number in the range 0000 to 0FFFF. You should not give other symbols names in this format, since doing so may produce a symbol with multiple definitions. In listings, the local name is shown in the macro definition, but the actual name is shown in expansions of macro calls.

Nonlocal labels may be used in a macro; but if the macro is used more than once, the same label will appear in both expansions, and **MASM** will display an error message, indicating that the file contains a symbol with multiple definitions. To avoid this problem, use only local labels (or redefinable equates) in macros.

Note

The **LOCAL** directive can only be used in macro definitions, and it must precede all other statements in the definition. If you try another statement (such as a comment instruction) before the **LOCAL** directive, an error will be generated.

■ Example

```
power       MACRO   factor,exponent     ;; Use for unsigned only
            LOCAL   again,gotzero       ;; Declare symbols for macro
            xor     dx,dx               ;; Clear DX
            mov     cx,exponent         ;; Exponent is count for loop
            mov     ax,1                ;; Multiply by 1 first time
            jcxz    gotzero             ;; Get out if exponent is zero
            mov     bx,factor
again:      mul     bx                  ;; Multiply until done
            loop    again
gotzero:
            ENDM
```

In this example, the **LOCAL** directive defines the local names `again` and `gotzero` as labels to be used within the `power` macro.

These local names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, `again` will be assigned the name `??0000` and `gotzero` will be assigned `??0001`. The second time through, `again` will be assigned `??0002` and `gotzero` will be assigned `??0003`, and so on.

11.2.4 Exiting from a Macro

Normally, **MASM** processes all the statements in a macro definition and then continues with the next statement after the macro call. However, you can use the **EXITM** directive to tell the assembler to terminate macro expansion before all the statements in the macro have been assembled.

When the **EXITM** directive is encountered, the assembler exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If **EXITM** is encountered in a nested macro or repeat block, **MASM** returns to expanding the outer block.

The **EXITM** directive is typically used with conditional directives to skip the last statements in a macro under specified conditions. Often macros using the **EXITM** directive contain repeat blocks or are called recursively.

■ Example

```
allocate    MACRO   times       ; Macro definition
x           =       0
            REPT    times       ;; Repeat up to 256 times
            IF      x GT 0FFh   ;; Is x > 255 yet?
            EXITM               ;; If so, quit
            ELSE
            DB      x           ;; Else allocate x
            ENDIF
x           =       x + 1       ;; Increment x
            ENDM
            ENDM
```

This example defines a macro that allocates a variable amount of data, but no more than 255 bytes. The macro contains an **IFE** directive that checks the expression `x - 0FFh`. When the value of this expression is true (x–255 = 0), the **EXITM** directive is processed and expansion of the macro stops.

11.3 Defining Repeat Blocks

Repeat blocks are a special form of macro that allows you to create blocks of repeated statements. They differ from macros in that they are not named, and thus cannot be called. However, like macros, they can have parameters that are replaced by actual arguments during assembly. Macro operators, symbols declared with the **LOCAL** directive, and the **EXITM** directive can be used in repeat blocks. Like macros, repeat blocks are always terminated by an **ENDM** directive.

Repeat blocks are frequently placed in macros in order to repeat some of the statements in the macro. They can also be used independently, usually for declaring arrays with repeated data elements.

Repeat block are processed at assembly time and should not be confused with the **REP** instruction, which causes string instructions to be repeated at run time, as explained in Chapter 18, "Processing Strings."

Three different kinds of repeat blocks can be defined by using the **REPT**, **IRP**, and **IRPC** directives. The difference between them is in how the number of repetitions is specified.

11.3.1 The REPT Directive

The **REPT** directive is used to create repeat blocks in which the number of repetitions is specified with a numeric argument.

■ Syntax

REPT *expression*
statements
ENDM

The *expression* must evaluate to a numeric constant (a 16-bit unsigned number). It specifies the number of repetitions. Any valid assembler statements may be placed within the repeat block.

■ Example

```
alphabet    LABEL   BYTE
x           =       0           ;; Initialize
            REPT    26          ;; Specify 26 repetitions
            DB      'A' + x     ;; Allocate ASCII code for letter
x           =       x + 1       ;; Increment
            ENDM
```

This example repeats the equal-sign (=) and **DB** directives to initialize ASCII values for each uppercase letter of the alphabet.

11.3.2 The IRP Directive

The **IRP** directive is used to create repeat blocks in which the number of repetitions, as well as parameters for each repetition, are specified in a list of arguments.

■ Syntax

IRP *parameter*,<*argument*[,*argument*]...>
statements
ENDM

The assembler *statements* inside the block are repeated once for each *argument* in the list enclosed by angle brackets (< >). The *parameter* is a name for a placeholder to be replaced by the current argument. Each argument can be text, such as a symbol, string, or numeric constant. Any number of arguments can be given. If multiple arguments are given, they must be separated by commas. The angle brackets (< >) around the argument list are required. The *parameter* can be used any number of times in the *statements*.

When **MASM** encounters an **IRP** directive, it makes one copy of the statements for each argument in the enclosed list. While copying the statements, it substitutes the current argument for all occurrences of *parameter* in these statements. If a null argument (< >) is found in the list, the dummy name is replaced with a null value. If the argument list is empty, the **IRP** directive is ignored and no statements are copied.

■ Example

```
numbers     LABEL   BYTE
            IRP     x,<0,1,2,3,4,5,6,7,8,9>
            DB      10 DUP(x)
            ENDM
```

This example repeats the **DB** directive 10 times, allocating 10 bytes for each number in the list. The resulting statements create 100 bytes of data, starting with 10 zeros, followed by 10 ones, and so on.

11.3.3 The IRPC Directive

The **IRPC** directive is used to create repeat blocks in which the number of repetitions, as well as arguments for each repetition, is specified in a string.

■ Syntax

IRPC *parameter,string*
statements
ENDM

The assembler *statements* inside the block are repeated as many times as there are character in *string*. The *parameter* is a name for a placeholder to be replaced by the current character in *string*. The string can be any combination of letters, digits, and other characters. It should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. The *parameter* can be used any number of times in these statements.

When **MASM** encounters an **IRPC** directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of *parameter* in these statements.

■ Example 1

```
ten         LABEL   BYTE
            IRPC    x,0123456789
            DB      x
            ENDM
```

Example 1 repeats the **DB** directive 10 times, once for each character in the string `0123456789`. The resulting statements create 10 bytes of data having the values 0–9.

■ **Example 2**

```
IRPC    letter,ABCDEFGHIJKLMNOPQRSTUVWXYZ
DB      '&letter'           ; Allocate uppercase letter
DB      '&letter'+20h       ; Allocate lowercase letter
DB      '&letter'-40h       ; Allocate number of letter
ENDM
```

Example 2 allocates the ASCII codes for uppercase, lowercase, and numeric versions of each letter in the string. Notice that the substitute operator (&) is required so that `letter` will be treated as an argument rather than a string. See Section 11.4.1, "Substitute Operator," for more information.

11.4 Using Macro Operators

Macro and conditional directives use the following special set of macro operators:

Operator	Definition
&	Substitute operator
< >	Literal-text operator
!	Literal-character operator
%	Expression operator
;;	Macro comment

When used in a macro definition, a macro call, a repeat block, or as the argument of a conditional-assembly directive, these operators carry out special control operations, such as text substitution.

11.4.1 Substitute Operator

The substitute operator (&) forces **MASM** to replace a parameter with its corresponding actual argument value.

■ **Syntax**

&parameter

The substitute operator can be used when a parameter immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

■ Example

```
errgen      MACRO   y,x
            PUBLIC  err&y
err&y       DB      'Error &y: &x'
            ENDM
```

In the example, **MASM** replaces &x with the value of the argument passed to the macro errgen. If the macro is called with the statement

```
            errgen  5,<Unreadable disk>
```

the macro is expanded to

```
err5        DB      'Error 5: Unreadable disk'
```

Note

For complex, nested macros, you can use extra ampersands to delay the replacement of a parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with z to make sure its replacement occurs while the **IRP** directive is being processed:

```
alloc       MACRO   x
            IRP     z,<1,2,3>
x&&z        DB      z
            ENDM
            ENDM
```

In this example, the dummy parameter x is replaced immediately when the macro is called. The dummy parameter z, however, is not replaced until the **IRP** directive is processed. This means the dummy parameter is replaced as many times as there are numbers in the **IRP** parameter list. If the macro is called with

```
            alloc   var
```

the macro will be expanded as shown below:

```
var1    DB      1
var2    DB      2
var3    DB      3
```

11.4.2 Literal-Text Operator

The literal-text operator (< >) directs **MASM** to treat a list as a single string rather than as separate arguments.

■ Syntax

<text>

The *text* is considered a single literal element even if it contains commas, spaces, or tabs. The literal-text operator is most often used in macro calls and with the **IRP** directive to ensure that values in a parameter list are treated as a single parameter.

The literal-text operator can also be used to force **MASM** to treat special characters, such as the semicolon or the ampersand, literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

■ Example

```
work    1,2,3,4,5           ; Passes five parameters
                            ;   to "work"

work    <1,2,3,4,5>         ; Passes one five-element
                            ;   parameter to "work"
```

Note

When the **IRP** directive is used inside a macro definition and when the argument list of the **IRP** directive is also a parameter of the macro, you must use the literal-text operator (< >) to enclose the macro parameter.

For example, in the following macro definition, the parameter x is used as the argument list for the **IRP** directive:

```
init        MACRO   x
            IRP     y,<x>
            DB      y
            ENDM
            ENDM
```

If this macro is called with

```
init    <0,1,2,3,4,5,6,7,8,9>
```

the macro removes the angle brackets from the parameter so that it is expanded as `0,1,2,3,4,5,6,7,8,9`. The brackets inside the repeat block are necessary to put the angle brackets back on. The repeat block is then expanded as shown below:

```
IRP     y,<0,1,2,3,4,5,6,7,8,9>
DB      y
ENDM
```

11.4.3 Literal-Character Operator

The literal-character operator (!) forces the assembler to treat a specified character literally rather than as a symbol.

■ Syntax

!character

The literal-character operator is used with special characters such as the semicolon or ampersand when meaning of the special character must be suppressed. Using the literal-character operator is the same as enclosing a single character in brackets. For example, `!!` is the same as `<!>`.

■ Example

```
errgen      MACRO   y,x
            PUBLIC  err&y
err&y       DB      'Error &y: &x'
            ENDM
            .
            .
            .
            errgen  103,<Expression !> 255>
```

The example macro call is expanded to allocate the string `Error 103: Expression > 255`. Without the literal-character operator, the greater-than symbol would be interpreted as the end of the argument and an error would result.

11.4.4 Expression Operator

The expression operator (%) causes the assembler to treat the argument following the operator as an expression.

■ Syntax

%text

MASM computes the expression's value and replaces *text* with the result. The expression can be either a numeric expression or a text equate. Handling text equates with this operator is a new feature in Version 5.0. Previous versions handled numeric expressions only. If there are additional arguments after an argument that uses the expression operator, the additional arguments must be preceded by a comma, not a space or tab.

The expression operator is typically used in macro calls when the programmer needs to pass the result of an expression rather than the actual expression to a macro.

■ Example

```
printe      MACRO   exp,val
            IF2                         ;; On pass 2 only
            %OUT    exp = val           ;; Display expression and result
            ENDIF                       ;;   to screen
            ENDM

sym1        EQU     100
sym2        EQU     200
msg         EQU     <"Hello, World.">

            printe  <sym1 + sym2>,%(sym1 + sym2)
            printe  msg,%msg
```

In the first macro call, the text literal `sym1 + sym2 =` is passed to the parameter `exp`, and the result of the expression is passed to the parameter `val`. In the second macro call, the equate name `msg` is passed to the parameter `exp`, and the text of the equate is passed to the parameter `val`. As a result, **MASM** displays the following messages:

```
sym1 + sym2 = 300
msg = "Hello, World."
```

The **%OUT** directive, which sends a message to the screen, is described in Section 12.1, "Sending Messages to the Standard Output Device"; the **IF2**

directive is described in Section 10.1.2, "Testing the Pass with IF1 and IF2 Directives."

11.4.5 Macro Comments

A macro comment is any text in a macro definition that does not need to be copied in the macro expansion. A double semicolon (;;) is used to start a macro comment.

■ Syntax

;;text

All *text* following the double semicolon (;;) is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the **.LALL**, **.XALL**, and **.SALL** directives, as described in Section 12.2.3, "Controlling Page Breaks."

11.5 Using Recursive, Nested, and Redefined Macros

The concept of replacing macro names with predefined macro text is simple, but in practice it has many implications and potentially unexpected side effects. The following sections discuss advanced macro features (such as nesting, recursion, and redefinition) and point out some side effects of macros.

11.5.1 Using Recursion

Macro definitions can be recursive: that is, they can call themselves. Using recursive macros is one way of doing repeated operations. The macro does a task, and then calls itself to do the task again. The recursion is repeated until a specified condition is met.

■ **Example**

```
pushall     MACRO   reg1,reg2,reg3,reg4,reg5,reg6
            IFNB    <reg1>          ;; If parameter not blank
            push    reg1            ;;   push one register and repeat
            pushall reg2,reg3,reg4,reg5,reg6
            ENDIF
            ENDM
            .
            .
            .
pushall     ax,bx,si,ds
pushall     cs,es
```

In this example, the `pushall` macro repeatedly calls itself to push a register given in a parameter until no parameters are left to push. A variable number of parameters (up to six) can be given.

11.5.2 Nesting Macro Definitions

One macro can define another. **MASM** does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Using a macro to create similar macros can make maintenance easier. If you want to change all the macros, change the outer macro and it automatically changes the others.

■ **Example**

```
shifts      MACRO   opname              ; Define macro that defines macros
opname&s    MACRO   operand,rotates
            IF      rotates LE 4
            REPT    rotates
            opname  operand,1           ;; One at a time is faster
            ENDM                        ;;   for 4 or less on 8088/8086
            ELSE
            mov     cl,rotates          ;; Using CL is faster
            opname  operand,cl          ;;   for more than 4 on 8088/8086
            ENDIF
            ENDM
            ENDM

            shifts  ror                 ; Call macro
            shifts  rol                 ;   to new macros
            shifts  shr
            shifts  shl
            shifts  rcl
            shifts  rcr
            shifts  sal
            shifts  sar
            .
            .
            .
            shrs    ax,5                ; Call defined macros
            rols    bx,3
```

This macro, when called as shown, creates macros for multiple shifts with each of the shift and rotate instructions. All the macro names are identical except for the instruction. For example, the macro for the **SHR** instruction is called `shrs`; the macro for the **ROL** instruction is called `rols`. If you want to enhance the macros by doing more parameter checking, you can modify the original macro. Doing so will change the created macros automatically. This macro uses the substitute operator, as described in Section 11.4.1.

11.5.3 Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macro calls are expanded like any other macro call, but only when the outer macro is called.

■ Example

```
ex          MACRO   text,val    ; Inner macro definition
            IF2
            %OUT    The expression (&text) has the value: &val
            ENDIF
            ENDM

express     MACRO   expression ; Outer macro definition
            ex      <expression>,%(expression)
            ENDM
            .
            .
            .
            express <4 + 2 * 7 - 3 MOD 4>
```

The two sample macros enable you to print the result of a complex expression to the screen by using the **%OUT** directive, even though that directive expects text rather than an expression (see Section 12.1, "Sending Messages to the Standard Output Device"). Being able to see the value of an expression is convenient during debugging.

Both macros are necessary. The `express` macro calls the `ex` macro, using operators to pass the expression both as text and as the value of the expression. With the call in the example, the assembler sends the following line to the standard output:

```
The expression (4 + 2 * 7 - 3 MOD 4) has the value: 15
```

You could get the same output by using only the `ex` macro, but you would have to type the expression twice and supply the macro operators in

the correct places yourself. The `express` macro does this for you automatically. Notice that expressions containing spaces must still be enclosed in angle brackets. Section 11.4.2, "Literal-Text Operator," explains why.

11.5.4 Redefining Macros

Macros can be redefined. You do not need to purge the macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no statements or comments between the **ENDM** directive of the nested redefinition and the **ENDM** directive of the original macro.

■ Example

```
getasciiz   MACRO
            .DATA
max         DB      80
actual      DB      ?
tmpstr      DB      80 DUP (?)
            .CODE
            mov     ah,0Ah
            mov     dx,OFFSET max
            int     21h
            mov     bl,actual
            xor     bh,bh
            mov     tmpstr[bx],0
getasciiz   MACRO
            mov     ah,0Ah
            mov     dx,OFFSET max
            int     21h
            mov     bl,actual
            xor     bh,bh
            mov     tmpstr[bx],0
            ENDM
            ENDM
```

This macro allocates data space the first time it is called, and then redefines itself so that it doesn't try to reallocate the data on subsequent calls.

11.5.5 Avoiding Inadvertent Substitutions

MASM replaces all parameters when they occur with the corresponding argument, even if the substitution is inappropriate. For example, if you

use a register name such as **AX** or **BH** as a parameter, **MASM** replaces all occurrences of that name when it expands the macro. If the macro definition contains statements that use the register, not the parameter, the macro will be incorrectly expanded. **MASM** will not warn you about using reserved names as macro parameters.

MASM does give a warning if you use a reserved name as a macro name. You can ignore the warning, but be aware that the reserved name will no longer have its original meaning. For example, if you define a macro called ADD, the **ADD** instruction will no longer be available. Your ADD macro takes its place.

11.6 Managing Macros and Equates

Macros and equates are often kept in a separate file and read into the assembler source file at assembly time. In this way, libraries of related macros and equates can be used by many different source files.

The **INCLUDE** directive is used to read an include file into a source file. Memory can be saved by using the **PURGE** directive to delete the unneeded macros from memory.

11.6.1 Using Include Files

The **INCLUDE** directive inserts source code from a specified file into the source file from which the directive is given.

- **Syntax**

INCLUDE *filespec*

The *filespec* must specify an existing file containing valid assembler statements. When the assembler encounters an **INCLUDE** directive, it opens the specified source file and begins processing its statements. When all statements have been read, **MASM** continues with the statement immediately following the **INCLUDE** directive.

The *filespec* can be given either as a file name, or as a complete or relative file specification including drive or directory name.

If a complete or relative file specification is given, **MASM** looks for the include file only in the specified directory. If a file name is given without a directory or drive name, **MASM** looks for the file in the following order:

1. If paths are specified with the **/I** option, **MASM** looks for the include file in the specified directory or directories. See Section 2.4.6, "Getting Command-Line Help," for more information on the **/I** option.
2. **MASM** looks for the include file in the current directory.
3. If an **INCLUDE** environment variable is defined, **MASM** looks for the include file in the directory or directories specified in the environment variable.

Nested **INCLUDE** directives are allowed. **MASM** marks included statements with the letter "C" in assembly listings.

Directories can be specified in **INCLUDE** path names with either the backslash (\) or the forward slash (/). This is for XENIX compatibility.

Note

Any standard code can be placed in an include file. However, include files are usually used only for macros, equates, and standard segment definitions. Standard procedures are usually assembled into separate object files and linked with the main source modules. The CodeView debugger can debug code in multiple modules, but it cannot debug code in include files.

■ Examples

```
INCLUDE fileio.mac                      ; File name only; use with
                                        ;   /I or environment
INCLUDE b:\include\keybd.inc            ; Complete file specification

INCLUDE /usr/jons/include/stdio.mac     ; Path name in XENIX format

INCLUDE masm_inc\define.inc             ; Partial path name in DOS format
                                        ;   (relative to current directory)
```

11.6.2 Purging Macros from Memory

The **PURGE** directive can be used to delete a currently defined macro from memory.

■ Syntax

PURGE *macroname*[,*macroname*]...

Each *macroname* is deleted from memory when the directive is encountered at assembly time. Any subsequent call to that macro causes the assembler to generate an error.

The **PURGE** directive is intended to clear memory space no longer needed by a macro. If a macro has been used to redefine a reserved name, the reserved name is restored to its previous meaning.

The **PURGE** directive can be used to clear memory if a macro or group of macros is needed only for part of a source file.

It is not necessary to purge a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, a macro can purge itself as long as the **PURGE** directive is on the last line of the macro.

The **PURGE** directive works by redefining the macro to a null string. Therefore, calling a purged macro does not cause an error. The macro name is simply ignored.

■ Examples

```
GetStuff
PURGE   GetStuff
```

These examples call a macro and then purge it. You might need to purge macros in this way if your system does not have enough memory to keep all the macros needed for a source file in memory at the same time.

CHAPTER 12

CONTROLLING ASSEMBLY OUTPUT

12.1 Sending Messages to the Standard Output Device.....241

12.2 Controlling Page Format in Listings.....242
- 12.2.1 Setting the Listing Title.....242
- 12.2.2 Setting the Listing Subtitle.....243
- 12.2.3 Controlling Page Breaks.....243

12.3 Controlling the Contents of Listings.....245
- 12.3.1 Suppressing and Restoring Listing Output..245
- 12.3.2 Controlling Listing of Conditional Blocks ...246
- 12.3.3 Controlling Listing of Macros.....247

12.4 Controlling Cross-Reference Output.....249

MASM has two ways of communicating results of an assembly to the user. It can write information to a listing, cross-reference, or object file; or it can display messages to the standard output device (ordinarily the screen).

Both kinds of output can be controlled from the command line or from inside a source file. The command lines and options that affect information output are described in Chapter 2, "Using MASM." This chapter explains the directives that directly control output from inside source files.

12.1 Sending Messages to the Standard Output Device

The **%OUT** directive instructs the assembler to display text to the standard output device. This device is normally the screen, but you can also redirect the output to a file or other device (see Section 2.3,"Controlling Message Output").

■ Syntax

%OUT *text*

The *text* can be any line of ASCII characters. If you want to display multiple lines, you must use a separate **%OUT** directive for each line.

The directive is useful for displaying messages at specific points of a long assembly. It can be used inside conditional-assembly blocks to display messages when certain conditions are met.

The **%OUT** directive generates output for both assembly passes. The **IF1** and **IF2** directives can be used for control when the directive is processed. Macros that enable you to output the value of expressions are shown in Section 11.5.3, "Nesting Macros Calls."

■ Example

```
IF1
%OUT    First Pass - OK
ENDIF
```

This sample block could be placed at the end of a source file so that the message `First Pass - OK` would be displayed at the end of the first pass, but ignored on the second pass.

12.2 Controlling Page Format in Listings

MASM provides several directives for controlling the page format of listings. These directives include the following:

Directive	Action
TITLE	Sets title for listings
SUBTTL	Sets title for sections in listings
PAGE	Sets page length and width, and controls page and section breaks

12.2.1 Setting the Listing Title

The **TITLE** directive specifies a title to be used on each page of assembly listings.

■ Syntax

TITLE *text*

The *text* can be any combination of characters up to 60 in length. The title is printed flush left on the second line of each page of the listing.

If no **TITLE** directive is given, the title will be blank. No more than one **TITLE** directive per module is allowed.

■ Example

```
TITLE Graphics Routines
```

This example sets the listing title. A page heading that reflects this title is shown below:

```
Microsoft (R) Macro Assembler Version 5.00                  9/25/87 12:00:00
Graphics Routines                                           Page     1-2
```

12.2.2 Setting the Listing Subtitle

The **SUBTTL** directive specifies the subtitle used on each page of assembly listings.

■ Syntax

SUBTTL *text*

The *text* can be any combination of characters up to 60 in length. The subtitle is printed flush left on the third line of the listing pages.

If no **SUBTTL** directive is used, or if no *text* is given for a **SUBTTL** directive, the subtitle line is left blank.

Any number of **SUBTTL** directives can be given in a program. Each new directive replaces the current subtitle with the new *text*. **SUBTTL** directives are often used just before a **PAGE +** statement, which creates a new section (see Section 12.2.3, "Controlling Page Breaks").

■ Example

```
SUBTTL Point Plotting Procedure
PAGE   +
```

The example above creates a section title and then creates a page break and a new section. A page heading that reflects this title is shown below:

```
Microsoft (R) Macro Assembler Version 5.00                  9/25/87 12:00:00
Graphics Routines                                           Page     3-1
Point Plotting Procedure
```

12.2.3 Controlling Page Breaks

The **PAGE** directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break in the listing.

■ Syntax

PAGE [[*length*]],*width*]]
PAGE +

If *length* and *width* are specified, the **PAGE** directive sets the maximum number of lines per page to *length* and the maximum number of characters

per line to *width*. The *length* must be in the range of 10–255 lines. The default page length is 50 lines. The *width* must be in the range of 60–132 characters. The default page width is 80 characters. To specify *width* without changing the default *length*, use a comma before *width*.

If no argument is given, **PAGE** starts a new page in the program listing by copying a form-feed character to the file and generating new title and subtitle lines.

If a plus sign follows **PAGE**, a page break occurs, the section number is incremented, and the page number is reset to 1. Program-listing page numbers have the following format:

section-page

The *section* is the section number within the module, and *page* is the page number within the section. By default, section and page numbers begin with 1-1. The **SUBTTL** directive and the **PAGE** directive can be used together to start a new section with a new subtitle. See Section 12.2.2, "Setting the Listing Subtitle," for an example.

■ Example 1

```
PAGE
```

Example 1 creates a page break.

■ Example 2

```
PAGE 58,90
```

Example 2 sets the maximum page length to 58 lines and the maximum width to 90 characters.

■ Example 3

```
PAGE ,132
```

Example 3 sets the maximum width to 132 characters. The current page length (either the default of 50 or a previously set value) remains unchanged.

■ **Example 4**

```
PAGE +
```

Example 4 creates a page break, increments the current section number, and sets the page number to 1. For example, if the preceding page was 3-6, the new page would be 4-1.

12.3 Controlling the Contents of Listings

MASM provides several directives for controlling what text will be shown in listings. The directives that control the contents of listings are shown below:

Directive	Action
.LIST	Lists statements in program listing
.XLIST	Suppresses listing of statements
.LFCOND	Lists false-conditional blocks in program listing
.SFCOND	Suppresses false-conditional listing
.TFCOND	Toggles false-conditional listing
.LALL	Includes macro expansions in program listing
.SALL	Suppresses listing of macro expansions
.XALL	Excludes comments from macro listing

12.3.1 Suppressing and Restoring Listing Output

The **.LIST** and **.XLIST** directives specify which source lines are included in the program listing.

■ **Syntax**

.LIST
.XLIST

The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. The directives are typically used in pairs to prevent a particular section of a source file from being copied to the program listing.

The **.XLIST** directive overrides other listing directives such as **.SFCOND** or **.LALL**.

■ **Example**

```
.XLIST              ; Listing suspended here
.
.
.
.LIST               ; Listing resumes here
.
.
.
```

12.3.2 Controlling Listing of Conditional Blocks

The **.SFCOND**, **.LFCOND**, and **.TFCOND** directives control whether false-conditional blocks should be included in assembly listings.

■ **Syntax**

.SFCOND
.LFCOND
.TFCOND

The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose condition is false. The **.LFCOND** directive restores the listing of these blocks. Like **.LIST** and **.XLIST**, conditional-listing directives can be used to suppress listing of conditional blocks in sections of a program.

The **.TFCOND** directive toggles the current status of listing of conditional blocks. This directive can be used in conjunction with the **/X** option of the assembler. By default, conditional blocks are not listed on start-up. However, they will be listed on start-up if the **/X** option is given. This means that using **/X** reverses the meaning of the first **.TFCOND** directive in the source file. The **/X** option is discussed in Section 2.4.14, "Listing False Conditionals."

■ **Example**

```
test1       EQU     0           ; Defined to make all conditionals false
                                ; /X not used       /X used
            .TFCOND
            IFNDEF  test1       ; Listed            Not listed
test2       DB      128
            ENDIF
            .TFCOND
            IFNDEF  test1       ; Not listed        Listed
test3       DB      128
            ENDIF
            .SFCOND
            IFNDEF  test1       ; Not listed        Not listed
test4       DB      128
            ENDIF
            .LFCOND
            IFNDEF  test1       ; Listed            Listed
test5       DB      128
            ENDIF
```

In the example above, the listing status for the first two conditional blocks would be different, depending on whether the **/X** option was used. The blocks with **.SFCOND** and **.LFCOND** would not be affected by the **/X** option.

12.3.3 Controlling Listing of Macros

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the expanded macros calls. The assembler always lists the full macro definition. The directives only affect expansion of macro calls.

■ **Syntax**

.LALL
.XALL
.SALL

The **.LALL** directive causes **MASM** to list all the source statements in a macro expansion, including normal comments (preceded by a single semicolon) but not macro comments (preceded by a double semicolon).

The **.XALL** directive causes **MASM** to list only those source statements in a macro expansion that generate code or data. For instance, comments, equates, and segment definitions are ignored.

The **.SALL** directive causes **MASM** to suppress listing of all macro expansions. The listing shows the macro call, but not the source lines generated by the call.

The **.XALL** directive is in effect when **MASM** first begins execution.

■ Example

```
tryout      MACRO   param
                                ;;Macro comment
                                ; Normal comment
it          EQU     3           ; No code or data
            ASSUME  es:_DATA    ; No code or data
            DW      param       ; Generates data
            mov     ax,it       ; Generates code
            ENDM
            .
            .
            .
            .XALL
            tryout  6           ; Call with .LALL

            .XALL
            tryout  6           ; Call with .XALL

            .SALL
            tryout  6           ; Call with .SALL
```

The macro calls in the example generate the following listing lines:

```
                                .LALL
                                tryout  6           ; Call with .LALL
                          1                         ; Normal comment
= 0003                    1 it  EQU     3           ; No code or data
                          1     ASSUME  es:_TEXT    ; No code or data
0015  0006                1     DW      6           ; Generates data
0017  B8 0003             1     mov     ax,it       ; Generates code


                                .XALL
                                tryout  6           ; Call with .XALL
001A  0006                1     DW      6           ; Generates data
001C  B8 0003             1     mov     ax,it       ; Generates code


                                .SALL
                                tryout  6           ; Call with .SALL
```

Notice that the macro comment is never listed in macro expansions. Normal comments are listed only with the **.LALL** directive.

12.4 Controlling Cross-Reference Output

The **.CREF** and **.XCREF** directives control the generation of cross-references for the macro assembler's cross-reference file.

■ Syntax

.CREF
.XCREF [*name*[,*name*]...]

The **.XCREF** directive suppresses the generation of label, variable, and symbol cross-references. The **.CREF** directive restores generation of cross-references.

If *names* are specified with **.XCREF**, only the named labels, variables, or symbols will be suppressed. All other names will be cross-referenced. The named labels, variables, or symbols will also be omitted from the symbol table of the program listing.

■ Example

```
.XCREF              ; Suppress cross-referencing
.                   ;   of symbols in this block
.
.
.CREF               ; Restore cross-referencing
.                   ;   of symbols in this block
.
.
.XCREF test1,test2  ; Don't cross-reference test1 or test2
.                   ;   in this block
.
.
```

PART 3

USING INSTRUCTIONS

CHAPTERS

13 Understanding 8086-Family Processors 255

14 Using Addressing Modes 271

15 Loading, Storing, and Moving Data 287

16 Doing Arithmetic and Bit Manipulations 305

17 Controlling Program Flow 331

18 Processing Strings 363

19 Calculating with a Math Coprocessor 377

20 Controlling the Processor 409

PART 3 ◇ USING INSTRUCTIONS

Part 3 of the Programmer's Guide explains how to use instructions in assembly-language source code. Instructions define the code that will be executed by the processor at run time.

Chapters 13 and 14 describe overall concepts that apply to all instructions. Chapter 13 summarizes the 8086-family processors; it explains protection modes, tells how the processors address memory, and describes registers. Chapter 14 explains the addressing modes that can be used with instruction operands.

Chapters 15–20 describe the instructions themselves. The material is organized topically, with related instructions discussed together. The 8087-family coprocessors and their instructions are explained in Chapter 19.

CHAPTER 13

UNDERSTANDING 8086-FAMILY PROCESSORS

13.1 Using the 8086-Family Processors 257
13.1.1 Processor Differences 257
13.1.2 Real and Protected Modes 259

13.2 Segmented Addresses 260

13.3 Using 8086-Family Registers 261
13.3.1 Segment Registers 263
13.3.2 General-Purpose Registers 264
13.3.3 Other Registers 266
13.3.4 The Flags Register 266
13.3.5 8087-Family Registers 268

13.4 Using the 80386 Processor Under DOS 269

This chapter introduces the 8086-family of processors. It describes their segmented-memory structure and their registers. Differences between the chips in the family are also covered.

13.1 Using the 8086-Family Processors

The Intel Corporation manufactures the group of processors referred to in this manual as the 8086-family processors. The MS-DOS and PC-DOS operating systems are designed to work under these processors and to take advantage of their features. The processors have several features in common, as follows:

- Memory is organized by using a segmented architecture.
- The instruction set is upwardly compatible—all features available in the early versions of the processor are also available in the newer versions, but the new versions contain additional features not supported in the old versions.
- The register set is also upwardly compatible.

13.1.1 Processor Differences

The main 8086-family processors are discussed below:

Processor	Description
8088 and 8086	These processors work in real mode. They are designed to run a single process. No provision is made to protect one part of memory from actions occurring in another part of memory. The processor can address up to one megabyte of memory. Addresses specified in assembly language correspond to physical memory addresses. The 8088 uses an 8-bit data bus, and the 8086 uses a 16-bit data bus. This makes the 8086 somewhat faster. However, from the programming standpoint, the two processors are identical except that the 8086 will handle certain data more efficiently if you word-align it by using the **EVEN** or **ALIGN** directives (see Section 6.5, "Aligning Data").

80186	This processor is identical to the 8086 except that new instructions have been added and some old instructions have been optimized. It runs significantly faster than the 8086. (There is also an enhanced version of the 8088 called the 80188.)
80286	This processor has the added instructions and speed of the 80186. It can run in the real mode of the 8088 and 8086, but it also has an optional protected mode in which multiple processes can be run concurrently. Memory used by each process can be protected from other processes. In protected mode, the processor can address up to 16 megabytes of memory. However, when memory is accessed in protected mode, the addresses do not correspond to physical memory. Under protected-mode operating systems, the processor allocates and manages memory dynamically. Additional privileged instructions for initializing protected mode and controlling multiple processes are available.
80386	This is both a 16-bit and a 32-bit processor. It is fully compatible with the 80286; but at the system level, it implements many new features, including virtual memory, multiple 8086 processes, and addressing for up to four gigabytes of memory. This manual does not explain how to use these features. For the applications programmer running in DOS, the 80836 supports all the instructions of the 80286 and some additional instructions. It also allows limited use of 32-bit registers and addressing modes. Finally, the 80386 operates significantly faster than the 80286. Considerations for programming the 80386 under DOS are summarized in Section 13.4.
8087, 80287, and 80387	These are math coprocessors that work concurrently with the 8086-family processors. They do mathematical calculations faster and more accurately than can be done with the 8086-family processors. Although there are performance and technical differences between the three coprocessors, the main difference to the applications programmer is that the 80287 and 80387 can operate in protected mode. The 80387 also has several new instructions.

13.1.2 Real and Protected Modes

Real mode is the single-process mode used in current versions of DOS. Protected mode is the multiple-process mode used in Microsoft XENIX. It will also be used in OS/2, the planned multitasking version of DOS.

To the applications programmer, there is little difference between assembly-language programming in real or protected mode. Processes are managed at the system level by the operating system. The applications programmer does not deal with processes except when interfacing with the operating system.

This manual does not address issues of interfacing with multitasking operating systems. If you are using a multitasking system, you must use the documentation for that operating system. However, applications programmers should be aware of the following differences between real- and protected-mode programming:

- In protected mode, up to 16 megabytes of memory can be addressed (compared to one megabyte in real mode). This distinction may make a difference in the number and size of data structures created, but it should make no difference in the assembly-language syntax, since data is addressed in exactly the same way in either mode.
- In protected mode, segment registers contain segment selectors rather than actual segment values. The selectors must come from the operating system. They cannot be calculated by the program. Programming techniques that attempt to calculate segment values or address memory directly will not work.
- The planned multitasking version of DOS, OS/2, will use the Applications Program Interface (API) to access DOS functions. This system is different from the current DOS system of using interrupt 21h.
- Certain instructions that can be used normally in real mode are privileged instructions in protected-mode operating systems. These include **STI**, **CLI**, **IN**, and **OUT**. These instructions are still available at privilege levels normally used only by systems programmers.

Protected-mode operating systems, such as XENIX and OS/2, provide extended functions for doing the kinds of tasks that are currently done by using the restricted practices described above.

13.2 Segmented Addresses

When used with current versions of DOS, 8086-family processors can store addresses as 16-bit word values. Therefore, the maximum unsigned value that can be stored as an address is 65,635 (0FFFFh). Yet the processors are actually capable of accessing much larger addresses. The highest possible address is one megabyte (0FFFFFh) in real mode or 16 megabytes (0FFFFFFh) in protected mode.

Addresses larger than 65,535 bytes are specified by combining two segmented word addresses: a 16-bit segment and a 16-bit offset within the segment. A common syntax for showing segmented addresses is the *segment:offset* format. For example, an address with a segment of 053C2h and an offset of 0107Ah would be represented as `53C2:107A`. This method of specifying addresses can be used directly in most debuggers, but it is not legal in assembler source code.

In real mode, the address `53C2:107A` represents a physical 20-bit address. This address can be calculated by multiplying the *segment* portion of the address by 16 (10h), and then adding the *offset* portion, as shown below:

```
  53C20h      Segment times 10h
+ 107Ah       Offset
--------
  54C9Ah      Physical address
```

In protected mode, the address `53C2:107A` represents a movable address. The segment portion of the address is a selector assigned a physical address by the operating system. The applications programmer has no control (and needs none) over the physical address represented by the selector.

80386 Only

The 80386 processor supports 48-bit addresses consisting of a 16-bit segment selector and a 32-bit offset. This enables the processor to access addresses of up to four gigabytes per segment in protected mode. The processor can also run in modes compatible with the 16-bit real- and protected-mode addressing schemes of the other 8086-family processors.

Addresses cannot be represented directly in the *segment:offset* format in assembly language. Instead the *segment* portion of the address is specified symbolically, using a name assigned to the segment in the source code. The address represented by the symbol can then be assigned to one of the segment registers. Chapter 5, "Defining Segment Structure," describes the directives that assign symbols to segment addresses.

The *offset* portion of addresses can be specified in a number of ways, depending on the context. Directives that assign symbols to offsets are discussed in Chapter 4, "Writing Source Code."

In assembly-language programming, addresses can be near or far. A near address is simply the offset portion of the address. Any instruction that accesses a near address will assume that the segment address is the same as the current segment for the type of address being accessed (usually a code segment for code or a data segment for data).

A far address consists of both the segment and offset portions of the address. Far addresses can be accessed from any segment. Both the segment and offset must be provided for instructions that access far addresses. Far addresses are more flexible because they can be used for larger programs and larger data objects. However, near addresses are more efficient, since they produce smaller code and can be accessed more quickly.

13.3 Using 8086-Family Registers

Like most microprocessors, the 8086-family processors have special areas of memory called registers. Some registers control the behavior or status of the processor. Others are used as temporary storage places where data can be accessed and processed faster than if data were stored in regular memory.

All the 8086-family processors share the same set of 16-bit registers. Some registers can be accessed as two separate 8-bit registers. In the 80386, most registers can also be accessed as extended 32-bit registers.

Figure 13.1 shows the registers common to all the 8086-family processors. Each register and group of registers has its own special uses and limitations, as described in this section.

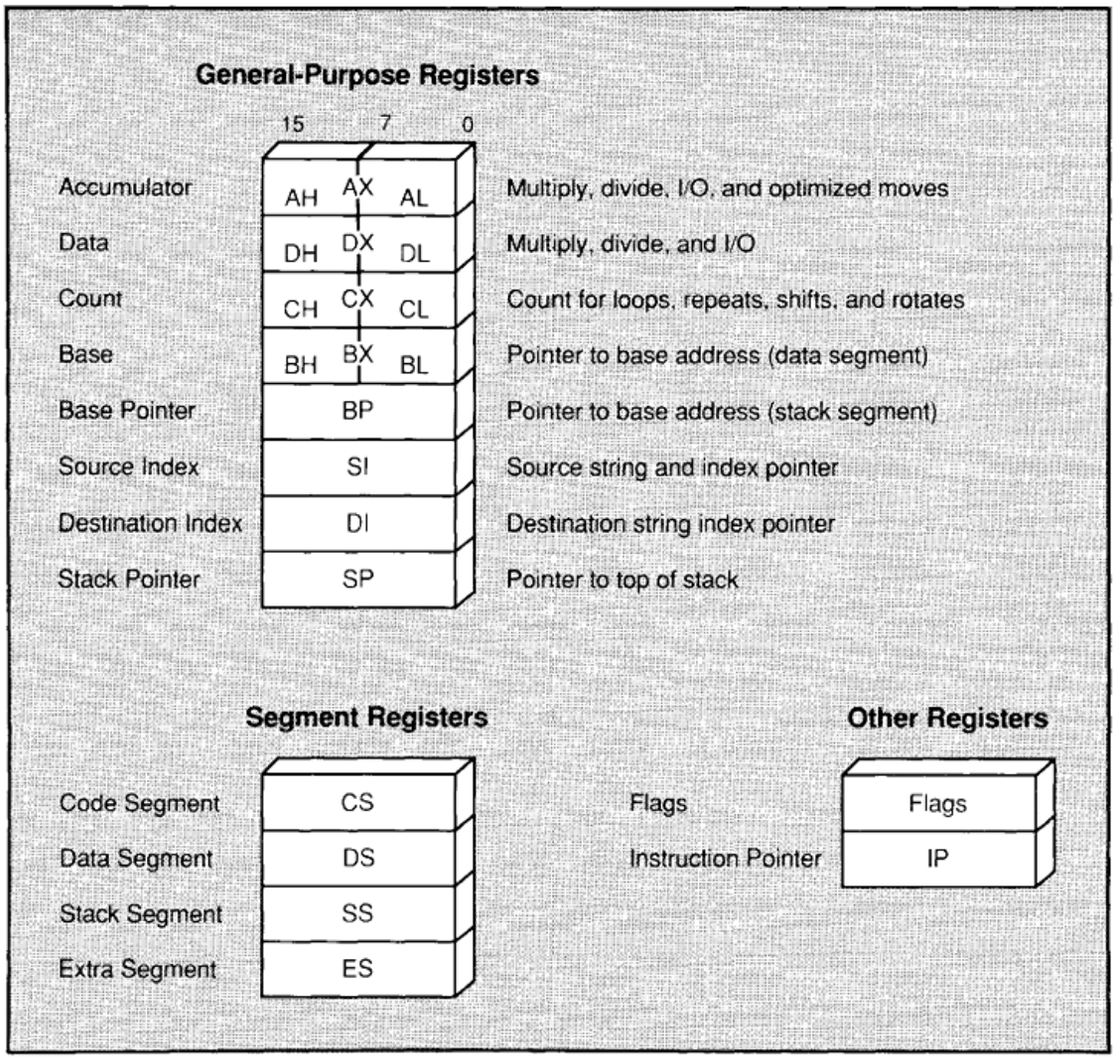


Figure 13.1 Register for 8088–80286 Processors

■ 80386 Only

The 80386 processor uses the same registers as the other processors in the 8086 family, but all except the segment registers can be extended to 32 bits. The extended registers begin with the letter **E**. For example, the 32-bit version of **AX** is **EAX**. The 80386 also has two additional segment registers, **FS** and **GS**. Figure 13.2 shows the extended registers of the 80386.

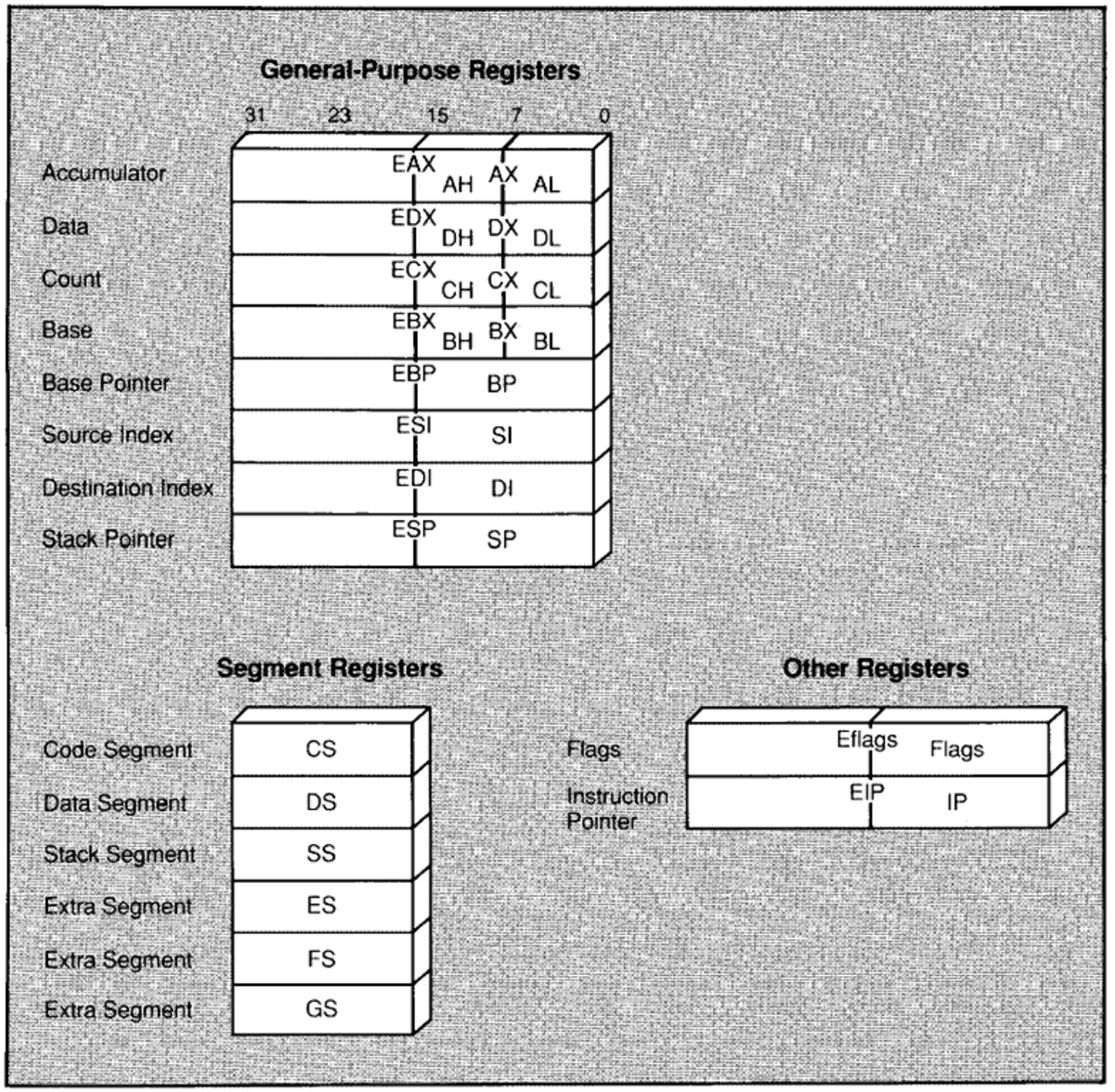


Figure 13.2 Extended Registers of 80386 Processor

13.3.1 Segment Registers

At run time, all addresses are relative to one of four segment registers: **CS**, **DS**, **SS**, or **ES**. These registers and the segments they correspond to are listed below:

Segment	Purpose
Code Segment (**CS**)	Addresses in the segment pointed to by this register contain the encoded instructions and operands specified by the program.

Data Segment (**DS**)	Addresses in the segment pointed to by this register normally contain data allocated by the program.
Stack Segment (**SS**)	Addresses in the segment pointed to by this register are available for instructions that store data on the program stack. A stack is an area of memory reserved for storing temporary data. See Section 15.4, "Transferring Data to and from the Stack," for information on using stacks.
Extra Segment (**ES**)	Addresses in the segment pointed to by this register are available for string instructions. An additional segment can also be stored in the **ES** register. The 80386 has two additional segments, **FS** and **GS**.

13.3.2 General-Purpose Registers

The **AX**, **DX**, **CX**, **BX**, **BP**, **SI**, and **DI** registers are 16-bit, general-purpose registers. They can be used to temporarily store data during processing. Data in registers can be accessed much more quickly than data in memory. Therefore, it is more efficient to keep the most frequently used values in registers.

Memory-to-memory operations are never allowed in 8086-family processors. As a result, data must often be moved into registers before doing calculations or other operations involving more than one variable.

Four of the general registers, **AX**, **DX**, **CX**, and **BX**, can be accessed as two 8-bit registers or as a single 16-bit register. The **AH**, **DH**, **CH**, **BH** registers represent the high-order 8 bits of the corresponding registers. Similarly, **AL**, **DL**, **CL**, and **BL** represent the low-order 8 bits of the registers. All the general registers can be extended to 32 bits on the 80386 by appending the letter **E**—**EAX**, **EDX**, **ECX**, and so on.

In addition to their general use for storing data, each of the general-purpose registers has special uses in certain situations. Specific uses for each register are listed below:

Register	Description
AX	The **AX** (Accumulator) register is most often used for storing temporary data. Many instructions are optimized so that they work slightly faster on data in the accumulator register than on data in other registers. With division instructions, the accumulator holds all or part of the dividend before the operation and the

quotient afterward. With multiplication instructions, the accumulator holds one of the factors before the operation and all or part of the result afterward. In I/O operations to and from ports, the accumulator holds the data being transferred.

DX The **DX** (Data) register is most often used for storing temporary data.

When dividing a doubleword value, **DX** holds the upper word of the dividend before the operation and the remainder afterward. When multiplying word values, **DX** holds the upper word of the doubleword result. In I/O operations to and from ports, **DX** holds the number of the port to be accessed.

CX The **CX** (Count) register must be used to hold the count for instructions that do looping or other repeated operations. These include the loop instructions, certain jump instructions, repeated string instructions, and shifts and rotates. This register can also be used for temporary data storage.

BX The **BX** (Base) register can be used as a pointer. For instance, it can point to the base of a data object (see Section 14.3.2, "Indirect Memory Operands"). This register can also be used for temporary data storage.

BP The **BP** (Base Pointer) register can be used for general data storage. It is more often used as a pointer. For instance, it is often used to point to the base of a stack frame. The Microsoft conventions for passing arguments to procedures have a specific use for **BP** as described in Section 17.4.3, "Passing Arguments on the Stack." The **SS** register is assumed as the segment register in operations using **BP**.

SI The **SI** (Source Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, **SI** is used to point to bytes or words within a source string.

DI The **DI** (Destination Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, **DI** is used to point to bytes or words within a destination string.

13.3.3 Other Registers

The 8086-family processors have two additional registers whose values are changed automatically by the processor.

Register	Description
SP	The **SP** (Stack Pointer) register points to the current location within the stack segment. Pushing a value onto the stack increases the value of **SP** by two; popping from the stack decreases the value of **SP** by two. Call instructions store the calling address on the stack and decrease **SP** accordingly; return instructions get the stored address and increase **SP**. With 80386 32-bit segments, **SP** is increased or decreased by four instead of two. Sections 15.4.2, "Using the Stack," and 17.4.3, "Passing Arguments on the Stack," discuss operation of the stack in more detail. **SP** is technically a general-purpose register that could be used in calculations or for temporary data storage. However, it should generally be used only for stack operations.
IP	The **IP** (Instruction Pointer) register always contains the address of the instruction about to be executed. The programmer cannot directly access or change the instruction pointer. However, instructions that control program flow (such as calls, jumps, loops, and interrupts) automatically change the instruction pointer.

13.3.4 The Flags Register

The flags register is a 16-bit register made up of bits that control various instructions and reflect the current status of the processor. In the 80386 processor, the flags register is extended to 32 bits. Some bits are undefined, so there are actually 9 flags for real mode, 11 flags (including a 2-bit flag) for 80286-protected mode, and 13 flags for the 80386. The extend flags register of the 80386 is sometimes called eflags.

Figure 13.3 shows the bits of the 32-bit flags register for the 808386. Only the lower word is used for the other 8086-family processors. The unmarked bits are reserved for processor use and should never be modified by the programmer.

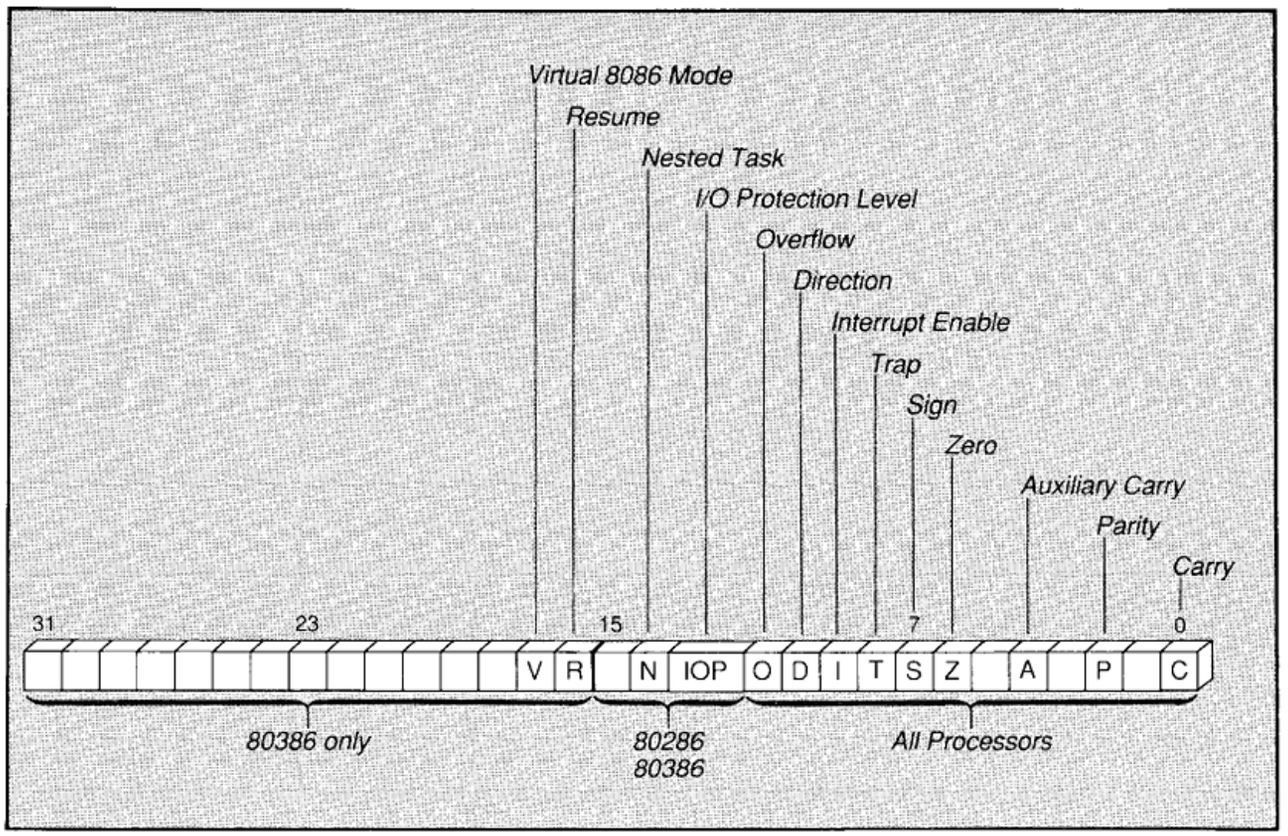


Figure 13.3 Flags for 8088–80386 Processors

The nine flags common to all 8086-family processors are summarized below, starting with the low-order flags. In these descriptions, the term "set" means the bit value is 1, and "cleared" means the bit value is 0.

Flag	Description
Carry	Is set if an operation generates a carry to or a borrow from a destination operand.
Parity	Is set if the low-order bits of the result of an operation contain an even number of set bits.
Auxiliary Carry	Is set if an operation generates a carry to or a borrow from the low-order four bits of an operand. This flag is used for binary-coded decimal arithmetic.
Zero	Is set if the result of an operation is 0.
Sign	Equal to the high-order bit of the result of an operation (0 is positive, 1 is negative).

Trap	If set, the processor generates a single-step interrupt after each instruction. A debugger program can use this feature to execute a program one instruction at a time.
Interrupt Enable	If set, interrupts will be recognized and acted on as they are received. The bit can be cleared to temporarily turn off interrupt processing.
Direction	Can be set to make string operations process down from high addresses to low addresses, or can be cleared to make string operations process up from low addresses to high addresses.
Overflow	Is set if the result of an operation is too large or small to fit in the destination operand.
I/O Protection Level	This 2-bit flag indicates the protection level for input and output. Managing the protection level is a systems task not described in this manual.
Nested Task	Controls chaining of interrupted and called tasks. Controlling tasks in protected mode is a systems task not described in this manual.
Resume	If set, debug exceptions are temporarily disabled. Using 80386 debug exceptions is a systems task not described in this manual.
Virtual 8086 Mode	If set, the processor is running an 8086-family real-mode program in a protected multitasking environment. If clear, the 80386 processor is in its normal mode. Running in virtual 8086 mode is a systems task not described in this manual.

13.3.5 8087-Family Registers

The 8087-family processors use a stack-based architecture to access up to eight 80-bit registers. See Chapter 19, "Calculating with a Math Coprocessor," for information on using 8087-family registers and instructions. The format of real numbers used by coprocessors is explained in Section 6.3.1.5, "Real-Number Variables."

13.4 Using the 80386 Processor Under DOS

Many of the added functions of the 80386 are not supported by versions of DOS available at release time for Version 5.0 of the Microsoft Macro Assembler. Although DOS runs on 80386 machines, it does not operate any differently (except faster) than on an 80286 machines. New features of the 80386, such as protected mode and 8086 virtual mode, are not supported by DOS. Since 32-bit segments are only available in protected mode, they cannot be used under DOS. Techniques for overcoming these limitations are beyond the scope of this manual.

Applications programmers can use some 80386 enhancements. The following features of the 80386 can be used under current versions of DOS. Note that using any of these features means your code will not run on machines that do not have an 80386 processor.

- You can use the new 80386 instructions (except for those that manage protected mode). New instructions include bit scan (**BSF** and **BFR**); bit test (**BT**, **BTC**, **BTR**, and **BTS**); move with sign and zero extend (**MOVSX** and **MOVZX**); set byte on condition (**SET***condition*); and double-precision shift (**SHLD** and **SHRD**).
- You can use 80286 instructions that have been enhanced to work with 32-bit registers. These include the integer-multiply instruction (**IMUL**); conversion instructions (**CWDE** and **CDQ**); string instructions (**CMPSD**, **LODSD**, **MOVSD**, **SCASD**, **STOSD**, **INSD**, **OUTSD**); and 32-bit stack enhancements (**PUSHAD**, **POPAD**, **PUSHFD**, **POPFD**, and **IRETD**).
- You can use 32-bit registers for calculations. For instance, you can add and subtract doubleword integers without using multiple registers, and you can do some multiplication and division operations on 64-bit integers.
- You can use 32-bit registers to point into 16-bit segments. In previous processors, only **BX**, **BP**, **DI**, and **SI** could be used as pointers in indirect memory operands. The 80386 has the same limitations on 16-bit registers, but allows any general-purpose 32-bit register to be a pointer in an indirect memory operand. If you use this technique, you must make sure that 32-bit registers used as pointers actually contain valid 16-bit addresses.

- If a program that uses 32-bit registers needs to execute while another program is running, it should save 32-bit registers at entry and restore them when finished. Under DOS, this applies only to device drivers and terminate-and-stay-resident programs that interrupt other programs. Use **PUSHAD** to save and **POPAD** to restore. Without these instructions, the interrupting program may change the upper half of 32-bit registers, causing errors when the interrupted program regains control.

Although significant, these new features fall short of using the full power that will be available under multiprocessing 80386 operating systems.

CHAPTER 14

USING ADDRESSING MODES

14.1 Using Immediate Operands 273

14.2 Using Register Operands 274

14.3 Using Memory Operands 276

14.3.1 Direct Memory Operands 276

14.3.2 Indirect Memory Operands 278

14.3.3 80386 Indirect Memory Operands 282

Instruction operands can be given in different forms called addressing modes. Addressing modes tell the processor how to calculate the actual value of an operand at run time.

The three kinds of addressing modes are immediate, register, and memory operands. Memory operands are further broken into two groups, direct and indirect memory operands.

The value of operands is calculated at assembly time for immediate operands, at load time for direct memory operands, and at run time for register operands and indirect memory operands.

Although two statements may be similar and their instruction mnemonic the same, **MASM** may actually assemble different code for an instruction when it is used with different addressing modes. For example, the statements

```
mov     ax,1
```

and

```
mov     ax,place[bx][di]
```

use the same instruction, but have different encoding, timing, and size. See the *Microsoft Macro Assembler Reference* for more information on the encoding, timing, and size of instructions.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be used, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be operated on and possibly changed by the instruction.

14.1 Using Immediate Operands

Immediate operands consist of constant numeric data that are known or calculated at assembly time. Immediate values are coded into the executable program and processed the same way each time the program is run.

Some instructions have limits on the size of immediate values (usually 8-, 16-, or 32-bit). String constants longer than two characters (four characters on the 80386) cannot be immediate data. They must be stored in memory before they can be processed by instructions.

Many instructions permit immediate data in the source (right) operand and either memory or register data in the destination (left) operand. The instruction combines or replaces the register or memory data with the immediate data in some way defined by the instruction. Examples of this type of instruction include **MOV**, **ADD**, **CMP**, and **XOR**.

A few instructions, such as **RET** and **INT**, take a single immediate operand.

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either register or direct memory so that there will be a place to store the result of the operation.

■ Examples

```
                .DATA
five            DB      5               ; Memory data
nine            EQU     9               ; Constant data

                .CODE
                .
                .
                .
; Source operand is immediate
                mov     bx,nine+3
                add     five,3
                or      bx,00100100b
                in      al,43h
                cmp     cx,200

; Only operand is immediate
                ret     6
                int     21h
```

14.2 Using Register Operands

Register operands consist of data stored in registers. Register-direct mode refers to using the actual value inside the register at the time the instruction is used. Registers can also be used indirectly to point to memory locations, as described in Section 14.3.2, "Indirect Memory Operands."

Most instructions allow register values in one or more operands. Some instructions can only be used with certain registers. Often instructions have shorter encoding (and faster operation) if the accumulator register

(**AX** or **AL**) is specified. Use of segment registers in operands is limited to a few instructions and special circumstances.

The registers shown in Table 14.1 can be used in register-direct mode.

Table 14.1

Register Operands

Register-Operand Type	Register Name			
8-bit high registers	**AH**	**BH**	**CH**	**DH**
8-bit low registers	**AL**	**BL**	**CL**	**DL**
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
32-bit general, pointer, and index[1]	**EAX**	**EBX**	**ECX**	**EDX**
16-bit pointer and index	**SP**	**BP**	**SI**	**DI**
32-bit general, pointer, and index[1]	**ESP**	**EBP**	**ESI**	**EDI**
16-bit segment	**CS**	**DS**	**SS**	**ES**
Additional 80386 segment[1]	**FS**	**GS**		

[1] Available only if the 80386 processor is enabled

Registers are discussed in more detail in Section 13.3. Limitations on register use for specific instructions are discussed in sections on the specific instructions throughout Part 3, "Using Instructions."

■ Examples

```
; Source and destination operands are register direct
            add     ax,bx
            mov     ds,ax
            xor     eax,ebx             ; 80386 only
            cmp     ah,bh

; Source operand is register direct
            and     stuff,dx
            sub     array[bx][si],ax

; Destination operand is register direct
            shl     ax,1
            cmp     cx,counter

; Only operand is register direct
            mul     bx
            pop     cx
            inc     ah
```

14.3 Using Memory Operands

Many instructions can work on data in memory. When a memory operand is given, the processor must calculate the address of the data to be processed. This address is called the "effective address." Calculation of the effective address depends on how the operand is specified, as explained below.

Note

Memory-to-memory operations are never allowed. These operations must be done indirectly by moving one of the memory values into a register before processing it.

14.3.1 Direct Memory Operands

A direct memory operand is a symbol that represents the address (segment and offset) of an instruction or data. The offset address represented by a direct memory operand is calculated at assembly time. The address of each operand relative to the start of the program is calculated at link time. The actual (or effective) address is calculated at load time.

Direct memory operands can be any constant or symbol representing an address. This includes labels, procedure names, variables, structure variables, record variables, or the value of the location counter.

The effective address is always relative to a segment register. The default segment register is **DS** for direct memory operands, but the default segment can be overridden with the segment-override operator (:), as explained in Section 9.2.3.

Direct memory operands are often specified as constant expressions by using the index operator. For example, the operand `table[4]` refers to the byte having an offset four bytes from the address of `table`. This expression is equivalent to `table+4`.

■ Example

```
            .DATA
stuff       DW      here
            .CODE
            .
            .
            .
            mov     ax,stuff          ; Load value at address "stuff"
                                      ;   (address of "here") into AX
            mov     bx,OFFSET stuff   ; Load address of "stuff"
                                      ;   into BX
            jmp     stuff             ; Jump to value of "stuff"
                                      ;   (which is address of "here")
            jmp     here              ; Jump to the address of "here"

            jmp     ax                ; Jump to AX (value of "stuff")

            jmp     [bx]              ; Jump to [BX] (value at address
            .                         ;   of "stuff")
            .
here:       .
```

This example illustrates the difference between memory operands that represent addresses and memory operands that represent the value at an address. Labels and variable names in the data segment (such as `stuff`) represent the value at an address. Code labels (such as `here`) represent the address itself. The four jump statements at the end of the example use different kinds of operands to transfer control to the same address.

Note

If the label is omitted from a direct memory operand used with a constant index, a segment must be specified. The offset of the operand is assumed to be the start of the specified segment plus the indexed offset. For example,

```
mov     ax,ds:[100h]
```

moves the value at address 100h in the data segment into the **AX** register. It is equivalent to

```
mov     ax,ds:100h
```

If the segment override is omitted, the constant (immediate) value of the operand is used rather than the value it points to. For example,

```
mov     ax,[100h]
```

moves the value 100h into the **AX** register. It is equivalent to the statement

```
mov     ax,100h
```

14.3.2 Indirect Memory Operands

Indirect memory operands enable you to use registers to point to values in memory. Since values in the registers can change at run time, you can use indirect memory operands to operate on data dynamically.

On all processors except the 80386, only four registers can be used in indirect mode (see Section 14.3.3, "80386 Indirect Memory Operands," for information on 80386 enhancements). **BX** and **BP** are called base registers; **DI** and **SI** are called index registers. The distinction between base and index registers is not always important. In many contexts, any of these registers can be thought of as the base or the index. In any case, an attempt to use any register other than these four in a statement that accesses memory indirectly results in an error.

You can use the base and index registers separately or in pairs, with or without specifying a displacement. A displacement can be either a constant or a direct memory. Several displacements can be given, but they are all added into a single displacement at assembly time. For example, in the statement

```
mov     ax,table[bx][di]+6
```

both `table` and `6` are displacements. **MASM** calculates the actual offset of `table` and at 6 to get the total displacement.

The modes in which registers can be used to specify indirect memory operands are shown in Table 14.2.

Table 14.2
Indirect Addressing Modes

Mode	Syntax	Description
Register indirect	[BX] [BP] [DI]	Effective address is contents of register
Based or indexed	[BX] *disp* *displacement*[BP] *displacement*[DI] *displacement*[SI]	Effective address is contents of register and *displacement*
Based indexed	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Effective address is contents of base register and contents of index register
Based indexed with displacement	*displacement*[BX][DI] *displacement*[BP][DI] *displacement*[BP][SI]	Effective address is contents of base register and contents of index registers and *displacement*

Register-indirect operands are typically used to point to a memory address within a segment. Based and indexed operands are used to point to a memory address relative to a table, a one-dimensional array, or a structure. Operands with multiple indexes are useful for pointing to memory locations in complex data structures such as multidimensional arrays.

The choice of which registers to use depends on the context of the statement. String instructions require that specific registers are used in specific situations, as explained in Chapter 18, "Processing Strings." With other instructions, base and index registers can often be used interchangeably, depending on which registers are available.

When calculating the effective address of an indirect operand, the processor uses **DS** as the default segment register if **BX** is used as a base register, or if no base register is specified. If **BP** is used anywhere in the operand, the default segment register is **SS**. The default segment can be overridden with the segment-override operator (:), as explained in Section 9.2.3 on the segment-override operator.

A common syntax for indirect memory operands is each register put within index operators ([]). The register or registers must always be within brackets, but a variety of alternate syntaxes is possible. Any operator that indicates addition can be used to combine the displacement and multiple registers. For example, the following statements are equivalent:

```
mov     ax,table[bx][di]
mov     ax,table[bx+di]
mov     ax,[table+bx+di]
mov     ax,[bx][di].table
mov     ax,[bx][di]+table
mov     ax,table[di][bx]
```

When using based-indexed modes, one of the registers must be a base register and the other an index register. The following statements are illegal:

```
mov     ax,table[bx][bp]    ; Illegal - two base registers
mov     ax,table[di][si]    ; Illegal - two index registers
```

Use of the index operator is explained in more detail in Section 9.2.1.3.

When an index or displacement points into an array, it must be scaled for the size of elements in the array. On all processors except the 80386, scaling must be done in separate statements (see Section 14.3.3, "80386 Indirect Memory Operands," for information on 80386 scaling). The scaling factor is 1 for bytes (no scaling necessary), 2 for words, 4 for doublewords, and 8 for quadwords. Since scaling factors (other than for bytes) are multiples of 2, they can usually be calculated quickly with the **SHL** instruction, as shown below:

```
shl     di,1        ; Scale DI for words (DI *2)

shl     di,1        ; Scale DI for doublewords (DI*4)
shl     di,1

shl     di,1        ; Scale DI for quadwords (DI*8)
shl     di,1
shl     di,1
```

Use of the **SHL** instruction for multiplication is described in more detail in Section 16.8.1, "Multiplying and Dividing by Constants."

■ Example 1

```
        add     dx,[bx]             ; Add the word contents of DS:BX
                                    ;   to the contents of DX
        mov     dl,[bp+6]           ; Load the byte contents
                                    ;   of SS:BP+6 into DL
        sub     dx,12[bx]           ; Subtract the word contents of
                                    ;   DS:12+BX from the contents of DX
        xor     red[bx],dx          ; XOR the contents of DX with
                                    ;   the contents of DS:red+BX
        and     dx,red[si]+3        ; AND the contents of DS:red+SI+3
                                    ;   with the contents of DX
        dec     BYTE PTR [bx][si]   ; Decrement the byte
                                    ;   at DS:BX+SI
        cmp     cx,here[bp][si]     ; Compare the contents of CX
                                    ;   to the contents of SS:here+BP+SI
        push    place[bx][di]+2     ; Save the contents of
                                    ;   DS:place+BX+DI+2 on the stack
        call    cs:table[bx]        ; Call the routine pointed to
                                    ;   by the contents of CS:table+bx
```

The statements in Example 1 illustrate how the various instructions can be used with indirect memory operands.

■ Example 2

```
scrnbuff    EQU     0B800h              ; CGA screen buffer (actual
                                        ;   value is hardware dependent)
            mov     ax,scrnbuff         ; Load address of screen buffer
            mov     es,ax               ;   into ES

            mov     ax,4                ; Push column 4 as third argument
            push    ax
            mov     ax,6                ; Push row 6 as second argument
            push    ax
            mov     ax,"z"              ; Push "z" as first argument
            push    ax
            call    show                ; Call the procedure
            add     sp,6                ; Restore stack
            .
            .
            .
show        PROC    NEAR
            push    bp                  ; Save BP
            mov     bp,sp               ;   and set up stack frame
            push    si                  ; Save SI (so procedure could
                                        ;   be called from C)

            mov     si,[bp+8]           ; Load column
            dec     si                  ; Adjust for zero
            shl     si,1                ; Scale for 2 bytes per character
            mov     bx,[bp+6]           ; Load row
            dec     bx                  ; Adjust for zero
            mov     ax,160              ; Multiply 160 bytes per line
            mul     bx                  ;   times current row
            mov     bx,ax               ; Put result in index

            mov     dl,BYTE PTR [bp+4]  ; Load character
            mov     es:[bx][si],dl      ; Put character in buffer

            pop     si                  ; Restore SI and BP
            pop     bp
            ret                         ; Return
show        ENDP
```

Example 2 illustrates two uses of indirect memory operands. Arguments are pushed onto the stack before calling a procedure. When the procedure is called, the arguments are removed using indirect memory operands.

The procedure writes a character to a screen buffer (a common technique with many computers and display adapters). The **BX** register points to the column position in the buffer; the **SI** register points to the row position. In this example, the **ES** register must contain the address of the screen buffer (this address varies for different hardware).

The procedure follows the calling conventions of Microsoft C and could be called directly from that language. Note that **SI** is saved and restored because the C compiler requires that it not be changed by a procedure.

Example 2 works on any processor. Section 14.3.3, "80386 Indirect Memory Operands," shows an enhanced version that uses 80386 instructions and addressing modes.

14.3.3 80386 Indirect Memory Operands

Instructions for the 80386 can be given in two modes, 16 bit and 32 bit. Understanding these modes is important, since indirect memory operands are different in each mode.

The 80386 instruction modes are controlled by the use type of the code segment in which the instructions are located. The mode is 16 bit if the use type is **USE16** or 32 bit if the use type is **USE32**. In 32-bit mode, an offset address can be up to four gigabytes. In 16-bit mode, an offset address can be up to 64K. The 16-bit mode of the 80386 is the same as the mode used by all the other 8086-family processors.

If the 80386 processor is enabled (with the **.386** directive), 32-bit general-purpose registers are always available. They can be used from 16-bit or 32-bit segments. When 32-bit registers are used, many of the limitations of 16-bit indirect memory modes do not apply. The following extensions are available when 32-bit registers are used in indirect memory operands:

- There are fewer limitations on the registers that can be used as base and index registers. With other 8086-family processors, only **BX**, **BP**, **DI**, and **SI** registers can be used in indirect memory operands. With the 80386, any general-purpose 32-bit register can be used. The same register can even be used as both the base and the index. Several examples are shown below:

```
add   edx,[eax]              ; Add double
mov   dl,[esp+10]            ; Add byte from stack
dec   WORD PTR [edx][eax]    ; Decrement word
cmp   cx,array[eax][eax]     ; Compare word from array
jmp   table[ecx]             ; Jump into pointer table
```

- The index register can have a scaling factor of 1, 2, 4, or 8. Any register except **ESP** can be the index register and can have a scaling factor. The scaling factor is specified by using the multiplication operator (*) adjacent to the register.

  Scaling can be used to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov   eax,darray[edx*4]      ; Load double of double array
mov   eax,[esi*8][edi]       ; Load double of quad array
mov   ax,wtbl[ecx+2][edx*2]  ; Load word of word array
```

- The default segment register is **SS** if the base register is **EBP** or **ESP**; it is **DS** for all other the base registers. If two registers are used, only one can have a scaling factor and it is defined to be the index register. The other register is the base. If scaling is not used, the first register is the base. If one register is used, it is the base, regardless of scaling. The following examples illustrate how to determine the base register:

```
mov   eax,[edx][ebp*4]  ; EDX base (not scaled) - DS segment
mov   eax,[edx*1][ebp]  ; EBP base (not scaled) - SS segment
mov   eax,[edx][ebp]    ; EDX base (first) - DS segment
mov   eax,[ebp][edx]    ; EBP base (first) - SS segment
mov   eax,[ebp*2]       ; EBP base (only) - SS segment
```

Statements can mix 16- and 32-bit registers. However, it is important to understand the implications of these statements. For example, the following statement is legal for either 16- or 32-bit segments:

```
mov     eax,[bx]
```

This moves the 32-bit value pointed to by **BX** into the **EAX** register. Although **BX** is a 16-bit pointer, it may still point into a 32-bit segment. However, the following statement is never legal:

```
mov     eax,[cx]
```

The **CX** register may not be used as a 16-bit pointer (although **ECX** may be used as a 32-bit pointer).

The following statement is also legal in either mode:

```
mov     bx,[eax]
```

This moves the 16-bit value pointed to by **EAX** into the **BX** register. This works fine in 32-bit mode; but in 16-bit mode, a 32-bit pointer moved into a 16-bit segment may cause problems. If **EAX** contains a 16-bit value (the

top half of the 32-bit register is 0), then the statement works. However, if the top half of the **EAX** register is not 0, the processor may generate an error.

Warning

It is possible to use both 16-bit and 32-bit modes in the same program by defining separate code segments for the two modes. However, this is a complex technique that involves special calculations to account for the differences between the two modes. Combining modes is generally done only in systems programming and is beyond the scope of this manual.

■ Example

```
            .MODEL  small               ; .MODEL preceeds .386
            .386                        ;   to make 16-bit segments

scrnbuff    EQU     OB800h              ; CGA screen buffer (actual
                                        ;   value is hardware dependent)
            .CODE
            .
            .
            .
            mov     ax,scrnbuff         ; Load address of screen buffer
            mov     es,ax               ;   into ES

            push    4                   ; Push column 4 as third argument
            push    6                   ; Push line 6 as second argument
            push    "z"                 ; Push "z" as first argument
            call    show                ; Call the procedure
            add     sp,6                ; Restore stack
            .
            .
            .
show        PROC    NEAR

            movzx   ebx,WORD PTR [esp+6]; Load column
            dec     ebx                 ; Adjust for zero
            movzx   eax,WORD PTR [esp+4]; Load row
            dec     eax                 ; Adjust for zero
            imul    eax,160             ; Multiply 160 bytes per line

            mov     dl,[esp+2]          ; Load character
            mov     es:[eax][ebx*2],dl  ; Put character in buffer

            ret                         ; Return
show        ENDP
```

This example is the same as the one in Section 14.3.2, "Indirect Memory Operands," except that it uses enhanced 80386 instructions and addressing modes to make the code shorter and more efficient. Note the following differences:

- Since **ESP** can be used as a base register, stack registers can be accessed directly without the stack setup required by previous processors. This assumes that **ESP** does not change inside the procedure.
- Values are loaded and zero-extended in one step by using the **MOVZX** instruction (see Section 15.2.3, "Moving and Extending Values").
- **EBX** is used with scaling. In the previous example, scaling had to be done with a separate instruction.
- **EAX** and **EBX** are used instead of **BX** and **SI**. This saves some register swapping, since **EAX** can be used both for the result of the multiplication operation and as a base register.
- Immediate operands are used with the **PUSH** and **IMUL** instructions (described in Sections 15.4.1, "Pushing and Popping," and 16.3, "Multiplying, " respectively). These enhancements were implemented with the 80186 processor, but they are rarely used since most programs have to be able to run on the 8088 and 8086. Since 80836 programs can never run on the earlier processors, there is no reason not to use enhanced 80186 instructions.

CHAPTER 15

LOADING, STORING, AND MOVING DATA

15.1 Transferring Data 289
- 15.1.1 Copying Data 289
- 15.1.2 Exchanging Data 290
- 15.1.3 Looking Up Data 290
- 15.1.4 Transferring Flags 291

15.2 Converting between Data Sizes 292
- 15.2.1 Extending Signed Values 292
- 15.2.2 Extending Unsigned Values 294
- 15.2.3 Moving and Extending Values 294

15.3 Loading Pointers 295
- 15.3.1 Loading Near Pointers 295
- 15.3.2 Loading Far Pointers 296

15.4 Transferring Data to and from the Stack 298
- 15.4.1 Pushing and Popping 298
- 15.4.2 Using the Stack 301
- 15.4.3 Saving Flags on the Stack 301
- 15.4.4 Saving All Registers on the Stack 302

15.5 Transferring Data to and from Ports 303

The 8086-family processors provide several instructions for loading, storing, or moving various kinds of data. Among the types of transferable data are variables, pointers, and flags. Data can be moved to and from registers, memory, ports, and the stack. This chapter explains the instructions for moving data from one location to another.

15.1 Transferring Data

Moving data is one of the most common tasks in assembly-language programming. Data can be moved between registers or between memory and registers. Immediate data can be loaded into registers or into memory.

15.1.1 Copying Data

The **MOV** instruction is the most common method of moving data. This instruction can be thought of as a "copy" instruction, since it always copies the source operand to the destination operand. Immediately after a **MOV** instruction, the source and destination operands both contain the same value. The old value in the destination operand is destroyed.

■ Syntax

MOV {*register* | *memory*},{*register* | *memory* | *immediate*}

■ Example 1

```
mov     ax,7        ; Immediate to register
mov     mem,7       ; Immediate to memory direct
mov     mem[bx],7   ; Immediate to memory indirect

mov     mem,ds      ; Segment register to memory
mov     mem,ax      ; Register to memory direct
mov     mem[bx],ax  ; Register to memory indirect

mov     ax,mem      ; Memory direct to register
mov     ax,mem[bx]  ; Memory indirect to register
mov     ds,mem      ; Memory to segment register

mov     ax,bx       ; Register to register
mov     ds,ax       ; General register to segment register
mov     ax,ds       ; Segment register to general register
```

The statements in Example 1 illustrate each type of memory move that can be done with a single instruction. Example 2 illustrates several common types of moves that require two instructions.

■ **Example 2**

```
; Move immediate to segment register
            mov     ax,DGROUP   ; Load immediate to general register
            mov     ds,ax       ; Store general register to segment register

; Move memory to memory
            mov     ax,mem1     ; Load memory to general register
            mov     mem2,ax     ; Store general register to memory

; Move segment register to segment register
            mov     ax,ds       ; Load segment register to general register
            mov     es,ax       ; Store general register to segment register
```

15.1.2 Exchanging Data

The **XCHG** (Exchange) instruction exchanges the data in the source and destination operands. Data can be exchanged between registers or between registers and memory.

■ **Syntax**

XCHG {*register* | *memory*},{*register* | *memory*}

■ **Examples**

```
xchg    ax,bx       ; Put AX in BX and BX in AX
xchg    memory,ax   ; Put "memory" in AX and AX in "memory"
```

15.1.3 Looking Up Data

The **XLAT** (Translate) instruction is used to load data from a table in memory. The instruction is useful for translating bytes from one coding system to another.

■ **Syntax**

XLAT[**B**] [[*segment*:]*memory*]

The **BX** register must contain the address of the start of the table. By default the **DS** register contains the segment of the table, but a segment override can be used to specify a different segment. The operand need not be given except when specifying a segment override.

Before the **XLAT** instruction is called, the **AL** register should contain a value that points into the table (the start of the table is considered 0). After the instruction is called, **AL** will contain the table value pointed to. For example, if **AL** contains 7, the 8th byte of the table will be placed in **AL** register.

Note

For compatibility with Intel 80386 mnemonics, **MASM** recognizes **XLATB** as a synonym for **XLAT**. In the Intel syntax, **XLAT** requires an operand; **XLATB** does not allow one. **MASM** never requires an operand, but always allows one.

■ Example

```
                ; Table of Hexadecimal digits
hex             DB      "0123456789ABCDEF"
convert         DB      "You pressed the key with ASCII code "
key             DB      ?,?,"h",13,10,"$"
                .CODE
                .
                .
                .
                mov     ah,8                ; Get a key in AL
                int     21h                 ; Call DOS
                mov     bx,OFFSET hex       ; Load table address
                mov     ah,al               ; Save a copy in high byte
                and     al,00001111b        ; Mask out top character
                xlat                        ; Translate
                mov     key[1],al           ; Store the character
                mov     cl,12               ; Load shift count
                shr     ax,cl               ; Shift high character into position
                xlat                        ; Translate
                mov     key,al              ; Store the character
                mov     dx,OFFSET convert   ; Load message
                mov     ah,9                ; Display it
                int     21h                 ; Call DOS
```

This example looks up hexadecimal characters in a table in order to convert an 8-bit binary number to a string representing a hexadecimal number.

15.1.4 Transferring Flags

The 8086-family processors provide instructions for loading and storing flags in the **AH** register.

■ **Syntax**

LAHF
SAHF

The status of the lower byte of the flags register can be saved to the **AH** register with **LAHF** and then later restored with **SAHF**. If you need to save and restore the entire flags register, use **PUSHF** and **POPF**, as described in Section 15.4.3, "Saving Flags on the Stack."

SAHF is often used with a coprocessor to transfer coprocessor control flags to processor control flags. Section 19.6, "Controlling Program Flow," explains and illustrates this technique.

15.2 Converting between Data Sizes

Since moving data between registers of different sizes is illegal, you must take special steps if you need to extend a register value to a larger register or register pair.

The procedure is different for signed and unsigned values. The processor cannot tell the difference between signed and unsigned numbers; the programmer has to understand this difference and program accordingly.

15.2.1 Extending Signed Values

The **CBW** (Convert Byte to Word) and **CWD** (Convert Word to Doubleword) instructions are provided to sign-extend values. Sign-extending means copying the sign bit of the unextended operand to all bits of the extended operand.

■ **Syntax**

CBW
CWD

The **CBW** instruction converts an 8-bit signed value in **AL** to a 16-bit signed value in **AX**. The **CWD** instruction is similar except that it sign-extends a 16-bit value in **AX** to a 32-bit value in the **DX:AX** register pair. Both instructions work only on values in the accumulator register.

■ Example 1

```
            .DATA
mem8        DB      -5
mem16       DW      -5
            .CODE
            .
            .
            .
            mov     al,mem8     ; Load 8-bit -5 (FBh)
            cbw                 ; Convert to 16-bit -5 (FFFBh) in AX

            mov     ax,mem16    ; Load 16-bit -5 (FFFBh)
            cwd                 ; Convert to 32-bit -5 (FFFF:FFFBh)
                                ;   in DX:AX
```

■ 80386 Only

The 80386 processor provides additional conversion instructions for 32-bit signed values.

■ Syntax

CWDE
CDQ

The **CWDE** (Convert Word to Doubleword Extended) instruction converts a signed 16-bit value in **AX** to a signed 32-bit signed value in **EAX**. The **CDQ** (Convert Doubleword to Quadword) instruction converts a 32-bit signed value in **EAX** to a signed 64-bit value in the **EDX:EAX** register pair.

■ Example 2

```
            .DATA
mem16       DW      -5
mem32       DD      -5
            .CODE
            .
            .
            .
            mov     ax,mem16    ; Load 16-bit -5 (FFFBh)
            cwde                ; Convert to 32-bit -5 (FFFFFFFBh) in EAX
            mov     eax,mem32   ; Load 32-bit -5 (FFFFFFFBh)
            cdq                 ; Convert to 64-bit -5
                                ;   (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

15.2.2 Extending Unsigned Values

To extend unsigned numbers, set the value of the upper register to 0.

■ Example

```
            .DATA
mem8        DB      251
mem16       DB      251
            .CODE
            .
            .
            .
            mov     al,mem8     ; Load 251 (FBh) from 8-bit memory
            xor     ah,ah       ; Zero upper half (AH)

            mov     ax,mem16    ; Load 251 (FBh) from 16-bit memory
            xor     dx,dx       ; Zero upper half (DX)
```

15.2.3 Moving and Extending Values

■ 80386 Only

The 80386 processor provides instructions that move and extend a value to a larger data size in a single step. The same thing can be done in two steps with earlier processors, but the new 80386 instructions are faster.

■ Syntax

MOVSX *register*,{ *register* | *memory*}
MOVZX *register*,{ *register* | *memory*}

MOVSX moves a signed value into a register and sign-extends it.
MOVZX moves an unsigned value into a register and zero-extends it.

■ Example

```
; Enhanced 80386 instructions

            movzx   dx,bl       ; Load unsigned 8-bit value into
                                ;   16-bit register and zero extend

; Equivalent to these 80286 instructions

            mov     dl,bl       ; Load 8-bit unsigned value
            xor     dh,dh       ; Clear the top of register

; Enhanced 80386 instructions

            movsx   dx,bl       ; Load unsigned 8-bit value into
                                ;   16-bit register and sign extend

; Equivalent to these 80286 instructions

            mov     al,bl       ; Load 8-bit unsigned value to AL
            cbw                 ; Sign extend to AX
            mov     dx,ax       ; Copy to 16-bit register
```

15.3 Loading Pointers

The 8086-family processors provide several instructions for loading pointer values into registers or register pairs. They can be used to load either near or far pointers.

15.3.1 Loading Near Pointers

The **LEA** instruction loads a near pointer into a specified register.

■ Syntax

LEA *register,memory*

The destination register may be any general-purpose register. The source operand may be any memory operand. The effective address of the source operand is placed in the destination register.

The **LEA** instruction can be used to calculate the effective address of a direct memory operand, but this is usually not efficient, since the address of a direct memory operand is a constant known at assembly time. For example, the following statements have the same effect, but the second version is faster:

```
lea     dx,string         ; Load effective address - slow
mov     dx,OFFSET string  ; Load offset - fast
```

The **LEA** instruction is more useful for calculating the address of indirect memory operands:

```
lea     dx,string[si]     ; Load effective address
```

■ 80386 Only

Scaling of indirect memory operands gives the **LEA** instruction some interesting side effects with the 80386 processor. (Scaling is explained in Section 14.3.3, "80386 Indirect Memory Operands." By using a 32-bit value as both the index and the base register in an indirect memory operand, you can multiply by the constants 2, 3, 4, 5, 8, and 9 more quickly than you could by using the **MUL** instruction.

```
lea     ebx,[eax*2]         ; EBX = 2 * EAX
lea     ebx,[eax*2+eax]     ; EBX = 3 * EAX
lea     ebx,[eax*4]         ; EBX = 4 * EAX
lea     ebx,[eax*4+eax]     ; EBX = 5 * EAX
lea     ebx,[eax*8]         ; EBX = 8 * EAX
lea     ebx,[eax*8+eax]     ; EBX = 9 * EAX
```

Multiplication by constants can also sometimes be made faster by using shift instructions, as described in Section 16.8.1, "Multiplying and Dividing by Constants."

15.3.2 Loading Far Pointers

The **LDS** and **LES** instructions load far pointers.

■ Syntax

LDS *register,memory*
LES *register,memory*

The memory address being pointed to is specified in the source operand, and the register where the offset will be stored is specified in the destination operand.

The address must be stored in memory with the offset in the upper word and the segment in the lower word. The segment register where the segment will be stored is specified in the instruction name. For example, **LDS** puts the segment in **DS**, and **LES** puts the segment in **ES**. These instructions are often used with string instructions, as explained in Chapter 18, "Processing Strings."

■ Example

```
            .DATA
string      DB      "This is a string."
fpstring    DD      string              ; Far pointer to string
pointers    DD      100 DUP (?)
            .CODE
            .
            .
            .
            les     di,fpstring         ; Put address in ES:DI pair
            lds     si,pointers[bx]     ; Put address in DS:SI pair
```

■ 80386 Only

The 80386 processor has additional instructions for loading far pointers. These instructions are exactly like **LDS** and **LES**, except for the segment register in which they put the segment address.

■ Syntax

LSS *register,memory*
LFS *register,memory*
LGS *register,memory*

The **LSS**, **LFS**, and **LGS** instructions load the segment address into **SS**, **FS**, and **GS** respectively.

■ Example

```
            .386                        ; .386 first for 32-bit mode
            .MODEL  large
            .DATA
string      DB      "This is a string."
fpstring    DF      string              ; Far pointer to string
            .CODE
            .
            .
            .
            lgs     edi,fpstring        ; Put address in GS:EDI pair
```

15.4 Transferring Data to and from the Stack

A stack is an area of memory for storing temporary data. Unlike other segments in which data is stored starting from low memory, data on the stack is stored in reverse order starting from high memory.

Initially, the stack is an uninitialized segment of a finite size. As data is added to the stack at run time, the stack grows downward from high memory to low memory. When items are removed from the stack, it shrinks upward from low memory to high memory.

The stack has several purposes in the 8086-family processors. The **CALL**, **INT**, **RET**, and **IRET** instructions automatically use the stack to store the calling addresses of procedures and interrupts (see Sections 17.4, "Using Procedures," and 17.5, "Using Interrupts"). You can also use the **PUSH** and **POP** instructions and their variations to store values on the stack.

15.4.1 Pushing and Popping

In 8086-family processors, the **SP** (stack pointer) register always points to the current location in the stack. The **PUSH** and **POP** instructions use the **SP** register to keep track of the current position in the stack.

The values pointed to by the **BP** and **SP** registers are relative to the stack segment (**SS** register). The **BP** register is often used to point to the base of a frame of reference (a stack frame) within the stack.

■ Syntax

PUSH {*register* | *memory*}
POP {*register* | *memory*}
PUSH *immediate* (80186–80386 only)

The **PUSH** instruction is used to store a two-byte operand on the stack. The **POP** instruction is used to retrieve a previously pushed value. When a value is pushed onto the stack, the **SP** register is decreased by two. When a value is popped off the stack, the **SP** register is increased by two. Although the stack always contains word values, the **SP** register points to bytes. Thus **SP** changes in multiples of two. (In 80386 32-bit segments, four-byte values are pushed and **SP** changes in multiples of four.)

Note

The 8088 and 8086 processors differ from later Intel processors in how they push and pop the **SP** register. If you give the statement `push sp` with the 8088 or 8086, the word pushed will be the word in **SP** after the push operation. The same statement under the 80186, 80286, or 80386 processor pushes the word in **SP** before the push operation.

Figure 15.1 illustrates how pushes and pops change the **SP** register. Notice that the value pushed onto the stack remains in stack memory even after it has been popped. However, since the stack pointer is above it, the value is now unknown and may be overwritten the next time the stack is used.

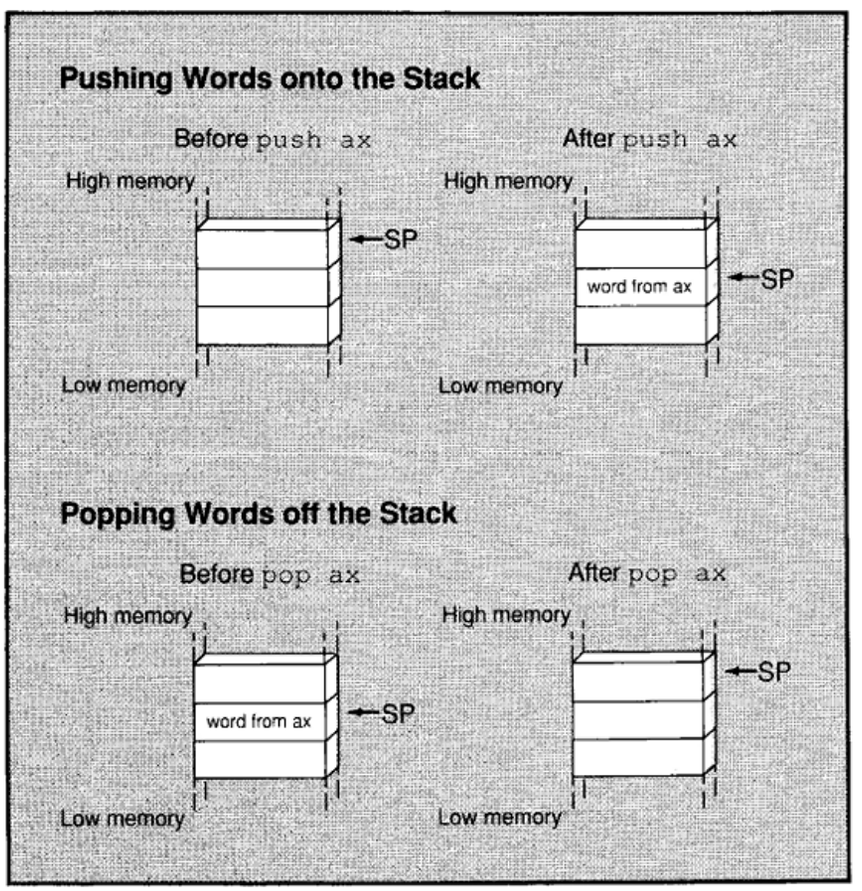


Figure 15.1 Stack Status after Pushes and Pops

The **PUSH** and **POP** instructions are almost always used in pairs. Words are popped off the stack in reverse order from the order in which they are pushed onto the stack. You should normally do the same number of pops as pushes to return the stack to its original status. However, it is possible to return the stack to its original status by subtracting the correct number of words from the **SP** register.

Values on the stack can be accessed by using indirect memory operands with **BP** as the base register.

■ Example

```
mov     bp,sp               ; Set stack frame
push    ax                  ; Push first;  SP = BP + 2
push    bx                  ; Push second; SP = BP + 4
push    cx                  ; Push third;  SP = BP + 6
.
.
.
mov     ax,[bp+6]           ; Put third in AX
mov     bx,[bp+4]           ; Put second in BX
mov     cx,[bp+2]           ; Put first in CX
.
.
.
sub     sp,6                ; Restore stack pointer
                            ;   two bytes per push
```

■ 80186/286/386 Only

Starting with the 80186, the **PUSH** instruction can be given with an immediate operand. For example, the following statement is legal on the 80186, 80286, and 80386 processors:

```
push    7           ; 3 clocks on 80286
```

This statement is faster than the following equivalent statements, which are required on the 8088 or 8086:

```
mov     ax,7        ; 2 clocks on 80286
push    ax          ; 3 clocks on 80286
```

■ 80386 Processor Only

When a **PUSH** or **POP** instruction is used in a 32-bit code segment (one with **USE32** use type), the value transferred is a four-byte value. A warning message will be generated if you try to push a 16-bit value in a 32-bit segment or a 32-bit value in a 16-bit segment.

15.4.2 Using the Stack

The stack can be used to store temporary data. For example, in the Microsoft calling convention, the stack is used to pass arguments to a procedure. The arguments are pushed onto the stack before the call. The procedure retrieves and uses them. Then the stack is restored to its original position at the end of the procedure. The stack can also be used to store variables that are local to a procedure. Both these techniques are discussed in Section 17.4.3, "Passing Arguments on the Stack."

Another common use of the stack is to store temporary data when there are no free registers available or when a particular register must hold more than one value. For example, the **CX** register usually holds the count for loops. If two loops are nested, the outer count is loaded into **CX** at the start. When the inner loop starts, the outer count is pushed onto the stack and the inner count loaded into **CX**. When the inner loop finishes, the originally count is popped back into **CX**.

■ Example

```
            mov     cx,10       ; Load outer loop counter
outer:      .
            .                   ; Start outer loop task
            .
            push    cx          ; Save outer loop value
            mov     cx,20       ; Load inner loop counter
inner:      .
            .                   ; Do inner loop task
            .
            loop    inner
            pop     cx          ; Restore outer loop counter
            .
            .                   ; Continue outer loop task
            .
            loop    outer
```

15.4.3 Saving Flags on the Stack

Flags can be pushed and popped onto the stack using the **PUSHF** and **POPF** instructions.

■ Syntax

PUSHF
POPF

These instructions are sometimes used to save the status of flags before a procedure call and then to restore the same status after the procedure. They can also be used within a procedure to save and restore the flag status of the caller.

■ **Example**

```
pushf
call    systask
popf
```

■ **80386 Only**

When used from a 32-bit code segment, the **PUSHF** and **POPF** instructions do not automatically transfer 32-bit values. You must append the letter **D** (for doubleword) to the instruction name. Thus the 32-bit versions of these instructions are **PUSHFD** and **POPFD**.

15.4.4 Saving All Registers on the Stack

■ **80186/286/386 Only**

Starting with the 80186 processor, the **PUSHA** and **POPA** instructions were implemented to push or pop all the general-purpose registers with one instruction.

■ **Syntax**

PUSHA
POPA

These instructions can be used to save the status of all registers before a procedure call and then to restore them after the return. Using **PUSHA** and **POPA** instructions is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

The registers are pushed in the following order: **AX**, **CX**, **DX**, **BX**, **SP**, **BP**, **SI**, and **DI**. The **SP** word pushed is the value before the first register is pushed. The registers are popped in the opposite order.

■ **Example**

```
pusha
call    systask
popa
```

■ **80386 Only**

When used from a 32-bit code segment, the **PUSHA** and **POPA** instructions do not automatically transfer 32-bit values. You must append the letter **D** (for doubleword) to the instruction name. Thus the 32-bit versions of these instructions are **PUSHAD** and **POPAD**.

15.5 Transferring Data to and from Ports

Ports are the gateways between hardware devices and the processor. Each port has a unique number through which it can be accessed. Ports can be used for low-level communication with devices such as disks, the video display, or the keyboard. The **OUT** instruction is used to send data to a port; the **IN** instruction receives data from a port.

■ **Syntax**

IN *accumulator*,{ *portnumber* | **DX**}
OUT { *portnumber* | **DX**},*accumulator*

When using the **IN** and **OUT** instructions, the number of the port can either be an 8-bit immediate value or the **DX** register. You must use **DX** for ports with a number higher than 256. The value to be received from the port must be in the accumulator register (**AX** for word values or **AL** for byte values).

When using the **IN** instruction, the number of the port is given as the source operand and the value to be sent to the port is the destination operand. When using the **OUT** instruction, the number of the port is given as the destination operand and the value to be sent to the port is the source operand.

In applications programming, most communication with hardware is done with DOS or BIOS calls. Ports are more often used in systems programming. Since systems programming is beyond the scope of this manual and since ports differ greatly depending on hardware, the **IN** and **OUT** instructions are not explained in detail here.

Note

Under protected-mode operating systems, **IN** and **OUT** are privileged instructions and can only be used in privileged mode.

■ Example

```
                                ; Actual values are hardware dependent
sound       EQU     61h         ; Port to chip that controls speaker
timer       EQU     42h         ; Port to chip that pulses speaker
on          EQU     00000011b   ; Bits 0 and 1 turn on speaker

            in      al,sound    ; Get current port setting
            or      al,on       ; Turn on speaker and connect timer
            out     sound,al    ; Put value back in port

            mov     al,50       ; Start at 50
sounder:    out     timer,al    ; Send byte to timer port...

            mov     cx,2000     ; Loop 2000 times to delay
hold:       loop    hold

            dec     al          ; Go down one step
            jnz     sounder     ; Repeat for each step

            in      al,sound    ; Get port value
            and     al,NOT on   ; Turn it back off
            out     sound,al    ; Put it back in port
```

This example creates a sound of ascending frequency on the IBM PC and IBM-compatible computers. The technique of making sound or the port values used may be different on other hardware.

■ 80186/286/386 Only

Starting with the 80186 processor, instructions were implemented to send strings of data to and from ports. The instructions are **INS**, **INSB**, **INSW**, **OUTS**, **OUTSB**, and **OUTSW**. The operation of these instructions is much like the operation of other string instructions. They are discussed in Section 18.7, "Transferring Strings to and from Ports."

CHAPTER 16

DOING ARITHMETIC AND BIT MANIPULATIONS

16.1 Adding 307
16.1.1 Adding Values Directly 307
16.1.2 Adding Values in Multiple Registers 309

16.2 Subtracting 309
16.2.1 Subtracting Values Directly 310
16.2.2 Subtracting with Values in Multiple Registers 311

16.3 Multiplying 312

16.4 Dividing 314

16.5 Calculating with Binary Coded Decimals 316
16.5.1 Unpacked BCD Numbers 317
16.5.2 Packed BCD Numbers 319

16.6 Doing Logical Bit Manipulations 320
16.6.1 AND Operations 321
16.6.2 OR Operations 322
16.6.3 XOR Operations 322
16.6.4 NOT Operations 323

16.7 Scanning for Set Bits 324

16.8 Shifting and Rotating Bits 325
16.8.1 Multiplying and Dividing by Constants 327
16.8.2 Moving Bits to the Least-Significant Position 329
16.8.3 Adjusting Masks 329
16.8.4 Shifting Multiword Values 329
16.8.5 Shifting Multiple Bits 330

The 8086-family processors provide instructions for doing calculations on byte, word, and doubleword values. Operations include addition, subtraction, multiplication, and division. You can also do calculations at the bit level. This includes the AND, OR, XOR, and NOT logical operations. Bits can also be shifted or rotated to the right or left.

This chapter tells you how to use the instructions that do calculations on numbers and bits.

16.1 Adding

The **ADD**, **ADC**, and **INC** instructions are used for adding and incrementing values.

■ Syntax

ADD {*register* | *memory*},{*register* | *memory* | *immediate*}
ADC {*register* | *memory*},{*register* | *memory* | *immediate*}
INC {*register* | *memory*}

These instructions can work directly on 8-bit or 16-bit values (32-bit values on the 80386). They can be also be used in combination to do calculations on values that are too large to be held in a single register (such as 32-bit values on the 80286 or 64-bit values on the 80386). When used with **AAA** and **DAA**, they can be used to do calculations on BCD numbers, as described in Section 16.5.

16.1.1 Adding Values Directly

The **ADD** and **INC** instructions are used for adding to values in registers or memory.

The **INC** instruction takes a single register or memory operand. The value of the operand is incremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed carries.

The **ADD** instruction adds values given in source and destination operands. The destination can be either a register or a memory operand. Its contents will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. Since memory-to-memory operations are never allowed, the source and destination operands can never both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both must be the same size.

An addition operation can be interpreted as addition of either signed numbers or unsigned numbers. It is the programmer's responsibility to decide how the addition should be interpreted and to take appropriate action if the sum is too large for the destination operand. When an addition overflows the possible range for signed numbers, the overflow flag is set. When an addition overflows the range for unsigned numbers, the carry flag is set.

There are two ways to take action on an overflow: you can use the **JO** or **JNO** instruction to direct program flow to or around instructions that handle the overflow (see Section 17.1.2.3, "Testing Bits and Jumping"). You can also use the **INTO** instruction to trigger the overflow interrupt (interrupt 4) if the overflow flag is set. This requires writing an interrupt handler for interrupt 4, since the DOS overflow routine simply returns without taking any action. Section 17.5.2, "Defining and Redefining Interrupt Routines," gives a sample of an overflow interrupt handler.

■ Examples

```
            .DATA
mem8        DB      39
            .CODE
            .
            .
            .                       ;                       unsigned  signed
            mov     al,26           ; Start with register     26      26
            inc     al              ; Increment                1       1
            add     al,76           ; Add immediate          + 76     76
                                    ;                         ----    ----
                                    ;                         103     103
            add     al,mem8         ; Add memory             + 39     39
                                    ;                         ----    ----
            mov     ah,al           ; Copy to AH              142    -114+overflow
            add     al,ah           ; Add register            142
                                    ;                         ----
                                    ;                           28+carry
```

This example shows 8-bit addition. When the sum exceeds 127, the overflow flag is set. A **JO** (Jump on Overflow) or **INTO** (Interrupt on Overflow) instruction at this point could transfer control to error-recovery statements. When the sum exceeds 255, the carry flag is set. A **JC** (Jump on Carry) instruction at this point could transfer control to error-recovery statements.

16.1.2 Adding Values in Multiple Registers

The **ADC** (Add with Carry) instruction makes it possible to add numbers larger than can be held in a single register.

The **ADC** instruction adds two numbers in the same fashion as the **ADD** instruction, except that the value of the carry flag is included in the addition. If a previous calculation has set the carry flag, then 1 will be added to the sum of the numbers. If the carry flag is not set, the **ADC** instruction has the same effect as the **ADD** instruction.

When adding numbers in multiple registers, the carry flag should be ignored for the least-significant portion, but taken into account for the more-significant portion. This can be done by using the **ADD** instruction for the least-significant portion and the **ADC** instruction for more-significant portions.

You can add and carry repeatedly inside a loop for calculations that require more than two registers. Use the **ADC** instruction in each iteration, but turn off the carry flag with the **CLC** (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration. You could also do the first add outside the loop.

■ **Example**

```
            .DATA
mem32       DD      316423
            .CODE
            .
            .
            .
            mov     ax,43981              ; Load immediate      43981
            xor     dx,dx                 ;   into DX:AX
            add     ax,WORD PTR mem32[0]  ; Add to both      + 316423
            adc     dx,WORD PTR mem32[2]  ;   memory words     ------
                                          ; Result in DX:AX    360404
```

16.2 Subtracting

The **SUB**, **SBB**, **DEC**, and **NEG** instructions are used for subtracting and decrementing values.

■ **Syntax**

SUB {*register* | *memory*},{*register* | *memory* | *immediate*}
SBB {*register* | *memory*},{*register* | *memory* | *immediate*}
DEC {*register* | *memory*}
NEG {*register* | *memory*}

These instructions can work directly on 8-bit or 16-bit values (32-bit values on the 80386). They can be also be used in combination to do calculations on values too large to be held in a single register (such as 32-bit values on the 80286 or 64-bit values on the 80386). When used with **AAA** and **DAA**, they can used to do calculations on BCD numbers, as described in Section 16.5.

16.2.1 Subtracting Values Directly

The **SUB** and **DEC** instructions are used for subtracting from values in registers or memory. A related instruction, **NEG** (Negate), reverses the sign of a number.

The **DEC** instruction takes a single register or memory operand. The value of the operand is decremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed borrows.

The **NEG** instruction takes a single register or memory operand. The sign of the value of the operand is reversed. The **NEG** instruction should only be used on signed numbers.

The **SUB** instruction subtracts the values given in the source operand from the value of the destination operand. The destination can be either a register or a memory operand. It will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. It will not be destroyed by the operation. Since memory-to-memory operations are never allowed, the source and destination operands cannot both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both must be the same size.

A subtraction operation can be interpreted as subtraction of either signed numbers or of unsigned numbers. It is the programmer's responsibility to decide how the subtraction should be interpreted and to take appropriate action if the result is too small for the destination operand. When a subtraction overflows the possible range for signed numbers, the carry flag is set. When a subtraction underflows the range for unsigned numbers (becomes negative), the sign flag is set.

■ **Example**

```
            .DATA
mem8        DB      122
            .CODE
            .
            .
            .                       ;                       signed  unsigned
            mov     al,95           ; Load register           95      95
            dec     al              ; Decrement             -   1   -   1
            sub     al,23           ; Subtract immediate    -  23   -  23
                                    ;                        ----    ----
                                    ;                          71      71
            sub     al,mem8         ; Subtract memory       - 122   - 122
                                    ;                        ----    ----
                                    ;                        - 51     205+sign

            mov     ah,119          ; Load register           119
            sub     al,ah           ;   and subtract        -- 51
                                    ;                        ----
                                    ;                          86+overflow
```

This example shows 8-bit subtraction. When the result goes below 0, the sign flag is set. A **JS** (Jump on Sign) instruction at this point could transfer control to error-recovery statements. When the result goes below –128, the carry flag is set. A **JC** (Jump on Carry) instruction at this point could transfer control to error-recovery statements.

16.2.2 Subtracting with Values in Multiple Registers

The **SBB** (Subtract with Borrow) instruction makes it possible to subtract from numbers larger than can be held in a single register.

The **SBB** instruction subtracts two numbers in the same fashion as the **SUB** instruction except that the value of the carry flag is included in the subtraction. If a previous calculation has set the carry flag, then 1 will be subtracted from the result. If the carry flag is not set, the **SBB** instruction has the same effect as the **SUB** instruction.

When subtracting numbers in multiple registers, the carry flag should be ignored for the least-significant portion, but taken into account for the more-significant portion. This can be done by using the **SUB** instruction for the least-significant portion and the **SBB** instruction for more-significant portions.

You can subtract and borrow repeatedly inside a loop for calculations that require more than two registers. Use the **SBB** instruction in each iteration, but turn off the carry flag with the **CLC** (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration. You could also do the first subtraction outside the loop.

■ Example

```
            .DATA
mem32a      DD      316423
mem32b      DD      156739
            .CODE
            .
            .
            .
            mov     ax,WORD PTR mem32a[0]  ; Load mem32          316423
            mov     dx,WORD PTR mem32a[2]  ;   into DX:AX
            sub     ax,WORD PTR mem32b[0]  ; Subtract low        156739
            sbb     dx,WORD PTR mem32b[2]  ;   then high         ------
                                           ; Result in DX:AX     159684
```

16.3 Multiplying

The **MUL** and **IMUL** instructions are used to multiply numbers. The **MUL** instruction should be used for unsigned numbers; the **IMUL** instruction should be used for signed numbers. This is the only difference between the two.

■ Syntax

MUL {*register* | *memory*}
IMUL {*register* | *memory*}

The multiply instructions require that one of the factors be in the accumulator register (**AL** for 8-bit numbers, **AX** for 16-bit numbers, or **EAX** for 32-bit numbers). This register is implied; it should not be specified in the source code. Its contents will be destroyed by the operation.

The other factor to be multiplied must be specified in a single register or memory operand. The operand will not be destroyed by the operation, unless it is **DX**, **AH**, or **AL**.

Note that multiplying two 8-bit numbers will produce a 16-bit number in **AX**. If the product is a 16-bit number, it will be placed in **AX** and the overflow and carry flags will be set.

Similarly, multiplying two 16-bit numbers will produce a 32-bit number in the **DX**:**AX** register pair. If the product is a 32-bit number, the most-significant bits will be in **AX**, the least-significant bits will be in **DX**, and the overflow and carry flags will be set. (The 80386 handles 64-bit products in the same way in the **EDX**:**EAX** register pair.)

Note

Multiplication is one of the slower operations on 8086-family processors (especially the 8086 and 8088). Multiplying by certain common constants is often faster when done by shifting bits (see Section 16.8.1, "Multiplying and Dividing by Constants") or by using 80386 scaling (see Section 15.3.1, "Loading Near Pointers").

■ Examples

```
            .DATA
mem16       DW      -30000
            .CODE
            .
            .
            .                       ; 8-bit unsigned multiply
            mov     al,23           ; Load AL                    23
            mov     bl,24           ; Load BL                  * 24
            mul     bl              ; Multiply BL              -----
                                    ; Product in AX             552
                                    ;   overflow and carry set

                                    ; 16-bit signed multiply
            mov     ax,50           ; Load AX                    50
                                    ;                         -30000
            imul    mem16           ; Multiply memory          -----
                                    ; Product in DX:AX     -1500000
                                    ;   overflow and carry set
```

■ 80186/286/386 Only

Starting with the 80186, the **IMUL** instruction has two additional syntaxes that allow for 16-bit multiples that produce a 16-bit product. (These instructions can be extended to 32 bits on the 80386.)

■ Syntax

IMUL *register16,immediate*
IMUL *register16,memory16,immediate*

You can specify a 16-bit immediate value as the source instruction and a word register as the destination operand. The product appears in the destination operand. The 16-bit result will be placed in the destination operand. If the product is too large to fit in 16 bits, the carry and overflow flags will be set. In this context, **IMUL** can be used for either signed or unsigned multiplication, since the 16-bit product is the same.

You can also specify three operands for **IMUL**. The first operand must be a 16-bit register operand, the second a 16-bit memory operand, and the third a 16-bit immediate operand. The second and third operands are multiplied and the product stored in the first operand.

With both these syntaxes, the carry and overflow flags will be set if the product is too large to fit in 16 bits. The **IMUL** instruction with multiple operands can be used for either signed or unsigned multiplication, since the 16-bit product is the same in either case. If you need to get a 32-bit result, you must use the single-operand version of **MUL** or **IMUL**.

■ Examples

```
imul    dx,456      ; Multiply DX times 456
imul    ax,[bx],6   ; Multiply the value pointed to by BX
                    ;   times 6 and put the result in AX
```

■ 80386 Only

On the 80386, the **IMUL** instruction has an additional instruction that allows multiplication of a register value by a register or memory value.

■ Syntax

IMUL *register*,{ *register* | *memory*}

The destination can be any 16-bit or 32-bit register. The source must be the same size as the destination.

■ Examples

```
imul    dx,ax       ; Multiply DX times AX
imul    ax,[bx]     ; Multiply AX by the value pointed to by BX
```

16.4 Dividing

The **DIV** and **IDIV** instructions are used to divide integers. Both a quotient and a remainder are returned. The **DIV** instruction should be used

for unsigned integers; the **IDIV** instruction should be used for signed integers. This is the only difference between the two.

■ Syntax

DIV {*register* | *memory*}
IDIV {*register* | *memory*}

To divide a 16-bit number by an 8-bit number, put the number to be divided (the dividend) in the **AX** register. The contents of this register will be destroyed by the operation. Specify the dividing number (the divisor) in any 8-bit memory or register operand (except **AL** or **AH**). This operand will not be changed by the operation. After the multiplication, the result (quotient) will be in **AL** and the remainder will be in **AH**.

To divide a 32-bit number by a 16-bit number, put the dividend in the **DX**:**AX** register pair. The most significant bits go in **AX**. The contents of these registers will be destroyed by the operation. Specify the divisor in any 16-bit memory or register operand (except **AX** or **DX**). This operand will not be changed by the operation. After the division, the quotient will be in **AX** and the remainder will be in **DX**. (The 80386 handles 64-bit division in the same way by using the **EDX**:**EAX** register pair.)

To divide a 16-bit number by a 16-bit number, you must first sign-extend or zero-extend (see Section 15.2, "Converting between Data Sizes") the dividend to 32 bits; then divide as described above. You cannot divide a 32-bit number by another 32-bit number (except on the 80386).

If division by zero is specified, or if the quotient exceeds the capacity of its register (**AL** or **AX**), the processor automatically generates an interrupt 0. By default, the program terminates and returns to DOS. This problem can be handled in two ways: you can check the divisor before division and go to an error routine if you can determine it to be invalid, or you can write your own interrupt routine to replace the processor's interrupt 0 routine. See Section 17.5 for more information in interrupts.

Note

Division is one of the slower operations on 8086-family processors (especially the 8086 and 8088). Dividing by common constants that are powers of two is often faster when done by shifting bits, as described in Section 16.8.1, "Multiplying and Dividing by Constants."

■ Examples

```
            .DATA

mem16       DW      -2000
mem32       DD      500000
            .CODE
            .
            .                               ; Divide 16-bit unsigned by 8-bit
            .
            mov     ax,700                  ; Load dividend          700
            mov     bl,36                   ; Load divisor      DIV   36
            div     bl                      ; Divide BL            -----
                                            ; Quotient in AL         19
                                            ; Remainder in AH           16

                                            ; Divide 32-bit signed by 16-bit

            mov     ax,WORD PTR mem32[0]    ; Load into DX:AX
            mov     dx,WORD PTR mem32[2]    ;                    500000
            idiv    mem16                   ;                DIV -2000
                                            ; Divide memory      ------
                                            ; Quotient in AX       -250
                                            ; Remainder in DX             0

                                            ; Divide 16-bit signed by 16-bit

            mov     ax,WORD PTR mem16       ; Load into AX         -2000
            cwd                             ; Extend to DX:AX
            mov     bx,-421                 ;                DIV -421
            idiv    bx                      ; Divide by BX         -----
                                            ; Quotient in AX           4
                                            ; Remainder in DX          -316
```

16.5 Calculating with Binary Coded Decimals

The 8086-family processors provide several instructions for adjusting BCD numbers. The BCD format is seldom used for applications programming in assembly language. Programmers who wish to use BCD numbers usually use a high-level language. However, BCD instructions are used to develop compilers, function libraries, and other systems tools.

Since systems programming is beyond the scope of this manual, this section provides only a brief overview of calculations on the two kinds of BCD numbers, unpacked and packed.

Note

Intel mnemonics use the term "ASCII" to refer to unpacked BCD numbers and "decimal" to refer to packed BCD numbers. Thus **AAA**

(ASCII Adjust for Addition) adjusts unpacked numbers, while **DAA** (Decimal Adjust for Addition) adjusts packed numbers.

16.5.1 Unpacked BCD Numbers

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower four bits of each byte. The 8086-family processors provide instructions for adjusting unpacked values with the four arithmetic operations—addition, subtraction, multiplication, and division.

To do arithmetic on unpacked BCD numbers, you must do the 8-bit arithmetic calculations on each digit separately. The result should always be in the **AL** register. After each operation, use the corresponding BCD instruction to adjust the result. The ASCII adjust instructions do not take an operand. They always work on the value in the **AL** register.

When a calculation using two one-digit values produces a two-digit result, the ASCII adjust instructions put the first digit in **AL** and the second in **AH**. If the digit in **AL** needs to carry to or borrow from the digit in **AH**, the carry and auxiliary carry flags are set.

The four ASCII adjust instructions are described below:

Instruction	Description
AAA	Adjusts after an addition operation. For example, to add 9 and 3, put 9 in **AL** and 3 in **BL**. Then use the following lines to add them:

```
mov     ax,9    ; Load 9
mov     bx,3    ;   and 3 as unpacked BCD
add     al,bl   ; Add 09h and 03h to get 0Ch
aaa             ; Adjust 0Ch in AL to 02h,
                ;   increment AH to 01h, set carry
                ; Result 12 unpacked BCD in AX
```

AAS	Adjusts after a subtraction operation. For example, to subtract 4 from 3, put 3 in **AL** and 4 in **BL**. Then use the following lines to subtract them:

```
mov     ax,103h ; Load 13
mov     bx,4    ;   and 4 as unpacked BCD
sub     al,bl   ; Subtract 4 from 3 to get FFh (-1)
aas             ; Adjust 0FFh in AL to 9,
                ;   decrement AH to 0, set carry
                ; Result 9 unpacked BCD in AX
```

AAM	Adjusts after a multiplication operation. Always use **MUL**, not **IMUL**. For example, to multiply 9 times

3, put 9 in **AL** and 3 in **BL**. Then use the following lines to multiply them:

```
mov     ax,903h ; Load 9 and 3 as unpacked BCD
mul     ah      ; Multiply 9 and 3 to get 1Bh
aam             ; Adjust 1Bh in AL
                ;   to get 27 unpacked BCD in AX
```

AAD

Adjusts before a division operation. Unlike other BCD instructions, this one converts a BCD value to a binary value before the operation. After the operation, the quotient must still be adjusted by using **AAM**. For example, to divide 25 by 2, put 25 in **AX** in unpacked BCD format: 2 in **AH** and 5 in **AL**. Put 2 in **BL**. Then use the following lines to divide them:

```
mov     ax,205h ; Load 25
mov     bl,2    ;   and 2 as unpacked BCD
aad             ; Adjust 0205h in AX
                ;   to get 19h in AX
div     bl      ; Divide by 2 to get
                ;   quotient 0Ch in AL
                ;   remainder 1 in AH
aam             ; Adjust 0Ch in AL
                ;   to 12 unpacked BCD in AX
                ;   (remainder destroyed)
```

Notice that the remainder is lost. If you need the remainder, save it in another register before adjusting the quotient. Then move it back to **AL** and adjust if necessary.

Multidigit BCD numbers are usually processed in loops. Each digit is processed and adjusted in turn.

In addition to their use for processing unpacked BCD numbers, the ASCII adjust instructions can be used in routines that convert between different number bases.

■ Example

```
mov     al,79       ; Load 79 (04Fh)
aam                 ; Adjust to BCD (0709h)
add     ah,48       ; Adjust to ASCII characters
add     al,48       ;   (3739h)
mov     dx,ax       ; Copy to DX
xchg    dl,dh       ; Trade for most significant digit
mov     ah,2        ; DOS display character function
int     21h         ; Call DOS
mov     dl,dh       ; Load least significant digit
int     21h         ; Call DOS
```

The example converts an 8-bit binary number to hexadecimal and displays it on the screen. The routine could be enhanced to handle large numbers.

16.5.2 Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper four bits and one in the lower four bits. The 8086-family processors provide instructions for adjusting packed BCD numbers after addition and subtraction. You must write your own routines to adjust for multiplication and division.

To do arithmetic on packed BCD numbers, you must do the eight-bit arithmetic calculations on each byte separately. The result should always be in the **AL** register. After each operation, use the corresponding BCD instruction to adjust the result. The decimal adjust instructions do not take an operand. They always work on the value in the **AL** register.

Unlike the ASCII adjust instructions, the decimal adjust instructions never affect **AH**. The auxiliary carry flag is set if the digit in the lower four bits carries to or borrows from the digit in the upper four bits. The carry flag is set if the digit in the upper four bits needs to carry to or borrow from another byte.

The decimal adjust instructions are described below:

Instruction	Description
DAA	Adjusts after an addition operation. For example, to add 88 and 33, put 88 in **AL** and 33 in **BL** in packed BCD format. Then use the following lines to add them:

```
mov     ax,8833h;Load 88 and 33 as packed BCD
add     al,ah   ; Add 88 and 33 to get OBBh
daa             ; Adjust OBBh to 121 packed BCD:
                ;   1 in carry and 21 in AL
```

DAS	Adjusts after a subtraction operation. For example, to subtract 38 from 83, put 83 in **AL** and 38 in **BL** in packed BCD format. Then use the following lines to subtract them:

```
mov     ax,3883h;Load 83 and 38 as packed BCD
sub     al,ah   ; Subtract 38 from 83 to get 04Bh
das             ; Adjust 04Bh to 45 packed BCD:
                ;   0 in carry and 45 in AL
```

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.

16.6 Doing Logical Bit Manipulations

The logical instructions do Boolean operations on individual bits. The AND, OR, XOR, and NOT operations are supported by the 8086-family instructions.

AND compares two bits and sets the result if both bits are set. OR compares two bits and sets the result if either bit is set. XOR compares two bits and sets the result if the bits are different. NOT reverses a single bit. Table 16.1 shows a truth table for the logical operations.

Table 16.1

Values Returned by Logical Operations

X	Y	NOT X	X AND Y	X OR Y	X XOR Y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

The syntax of the **AND**, **OR**, and **XOR** instructions are the same. The only difference is the operation performed. For all instructions, the target value to be changed by the operation is placed in one operand. A mask showing the positions of bits to be changed is placed in the other operand. The format of the mask differs for each logical instruction. The destination operand can be register or memory. The source operand can be register, memory, or immediate. However, the source and destination operands cannot both be memory.

Either of the values can be in either operand. However, the source operand will be unchanged by the operation, while the destination operand will be destroyed by it. Your choice of operands depends on whether you want to save a copy of the mask or of the target value.

Note

The logical instructions should not be confused with the logical operators. They specify completely different behavior. The instructions control run-time bit calculations. The operators control assembly-time bit

calculations. Although the instructions and operators have the same name, the assembler can distinguish them from context.

16.6.1 AND Operations

The **AND** instruction does an AND operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ Syntax

AND {*register* | *memory*},{*register* | *memory* | *immediate*}

The **AND** instruction can be used to clear the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 0 for any bit positions you want to clear and 1 for any bit positions you want to remain unchanged.

■ Example 1

```
        mov     ax,035h     ; Load value                00110101
        and     ax,0FBh     ; Mask off bit 2        AND 11111011
                            ;                           --------
                            ; Value is now 31h          00110001
        and     ax,0F8h     ; Mask off bits 2,1,0   AND 11111000
                            ;                           --------
                            ; Value is now 30h          00110000
```

■ Example 2

```
        mov     ah,7        ; Get character without echo
        int     21h
        and     al,11011111b ; Convert to uppercase by clearing bit 5
        cmp     al,'Y'      ; Is it Y?
        je      yes         ; If so, do Yes stuff
        .                   ;   else do No stuff
        .
yes:    .
```

Example 2 illustrates how to use the **AND** instruction to convert a character to uppercase. If the character is already uppercase, the **AND** instruction has no effect, since bit 5 is always clear in uppercase letters. If the character is lowercase, clearing bit 5 converts it to uppercase.

16.6.2 OR Operations

The **OR** instruction does an OR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ Syntax

OR {*register* | *memory*},{*register* | *memory* | *immediate*}

The **OR** instruction can be used to set the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 1 for any bit positions you want to set and 0 for any bit positions you want to remain unchanged.

■ Example

```
mov     ax,035h     ; Move value to register      00110101
mov     ax,035h     ; Move value to register      00110101
or      ax,08h      ; Mask on bit 3           OR  00001000
                    ;                             --------
                    ; Value is now 3Dh            00111101
or      ax,07h      ; Mask on bits 2,1,0      OR  00000111
                    ;                             --------
                    ; Value is now 3Fh            00111111
```

Another common use for **OR** is to compare an operand to 0. For example:

```
or      bx,bx       ; Compare to 0
                    ;   2 bytes, 2 clocks on 8088
jg      positive    ; BX is positive
jl      negative    ; BX is negative
                    ; BX is zero
```

The first statement has the same effect as the following statement, but is faster and smaller:

```
cmp     bx,0        ; 3 bytes, 3 clocks on 8088
```

16.6.3 XOR Operations

The **XOR** (Exclusive OR) instruction does an XOR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ **Syntax**

XOR {*register* | *memory*},{*register* | *memory* | *immediate*}

The **XOR** instruction can be used to toggle the value of specific bits (reverse them from their current settings). To do this, put the target value in one operand and a mask of the bits you want to toggle in the other. The bits of the mask should be 1 for any bit positions you want to toggle and 0 for any bit positions you want to remain unchanged.

■ **Example**

```
mov     ax,035h     ; Move value to register      00110101
xor     ax,08h      ; Mask on bit 3           XOR 00001000
                    ;                             --------
                    ; Value is now 3Dh            00111101
xor     ax,07h      ; Mask on bits 2,1,0      XOR 00000111
                    ;                             --------
                    ; Value is now 3Ah            00111010
```

Another common use for the **XOR** instruction is to set a register to 0. For example:

```
xor     cx,cx       ; 2 bytes, 3 clocks on 8088
```

This sets the **CX** register to 0. When the identical operands are XORed, each bit cancels itself, producing 0. The statement

```
mov     cx,0        ; 3 bytes, 4 clocks on 8088
```

is the obvious way of doing this, but it is larger and slower. The statement

```
sub     cx,cx       ; 2 bytes, 3 clocks on 8088
```

is also smaller than the **MOV** version. The only advantage of using **MOV** is that it does not affect any flags.

16.6.4 NOT Operations

The **NOT** instruction does a NOT operation on the bits of a single operand. It is used to toggle the value of all bits at once.

■ **Syntax**

NOT {*register* | *memory*}

The **NOT** instruction is often used to reverse the sense of a bit mask from masking certain bits on to masking them off. Use the **NOT** instruction if

the value of the mask is not known until run time; use the NOT operator (see Section 9.2.1.5, "Bitwise Logical Operators") if the mask is a constant.

■ Example

```
            .DATA
masker      DB      00010000b  ; Value may change at run time
            .CODE
            .
            .
            .
            mov     ax,0D743h  ; Load 0D7h to AH; 43h to AL  01000011
            or      al,masker  ; Turn on bit 4 in AL      OR 00010000
                               ;                            --------
                               ; Result is 53h              01010011

            not     masker     ; Reverse sense of mask      11101111
            and     ah,masker  ; Turn off bit 4 in AH    AND 11010111
                               ;                            --------
                               ; Result is 0C7h             11000111
```

16.7 Scanning for Set Bits

■ 80386 Only

The 80386 processor has instructions for scanning bits to find the first or last set bit in a register value. These instructions can be used to find the position of a set bit in a mask or other value. They can also check to see if a register value is 0.

■ Syntax

BSF *register,{ register | memory}*
BSR *register,{ register | memory}*

The bit scan instructions work only on 16-bit or 32-bit registers. They cannot be used on memory operands or 8-bit registers. The source register contains the value to be scanned. The destination register should be the register where you want to store the position of the first or last set bit.

The **BSF** (Bit Scan Forward) instruction scans the bits of the source register starting with the 0 bit and working toward the most-significant bit. The **BSR** (Bit Scan Reverse) instruction scans the bits of the source register starting with the most-significant bit and working toward the 0 bit.

■ **Example**

```
            .DATA
widfield    EQU     200
bitfield    DD      widfield DUP (?)
            .CODE
            .
            .
            .
            cld
            push    ds                  ; Load segment of bitfield
            pop     es                  ;   into ES
            mov     cx,widfield         ; Load maximum count
            xor     eax,eax             ; Set search value to 0
            mov     di,OFFSET bitfield  ; Load bitfield address
            repe    scasd               ; Find first nonzero bit
            jecxz   none                ; If none found, get out
            sub     di,4                ; Point back to doubleword
            mov     eax,[di]            ; Else load first nonzero
            bsr     ecx,eax             ; Find first set bit
            .                           ; ECX now contains bit position
            .                           ; DI points to doubleword
none:       .
```

This example scans a large bit field. Starting at the beginning of the field, it finds the first nonzero doubleword. Then it finds the first set bit within the doubleword. See Chapter 18, "Processing Strings," for more information on the string instructions used in this example.

16.8 Shifting and Rotating Bits

The 8086-family processors provide a complete set of instructions for shifting and rotating bits. Bits can be moved right (toward the most-significant bits) or left (toward the 0 bit). Values shifted off the end of the operand go into the carry flag.

Shift instructions move bits a specified number of places to the right or left. The last bit in the direction of the shift goes into the carry flag, and the first bit is filled with 0 or with the previous value of the first bit.

Rotate instructions move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate is moved into the first bit position at the other end of the operand. With some variations, the carry bit is used as an additional bit of the operand.

Figure 16.1 illustrates the eight variations of shift and rotate instructions for 8-bit operands. Notice that **SHL** and **SAL** are exactly the same.

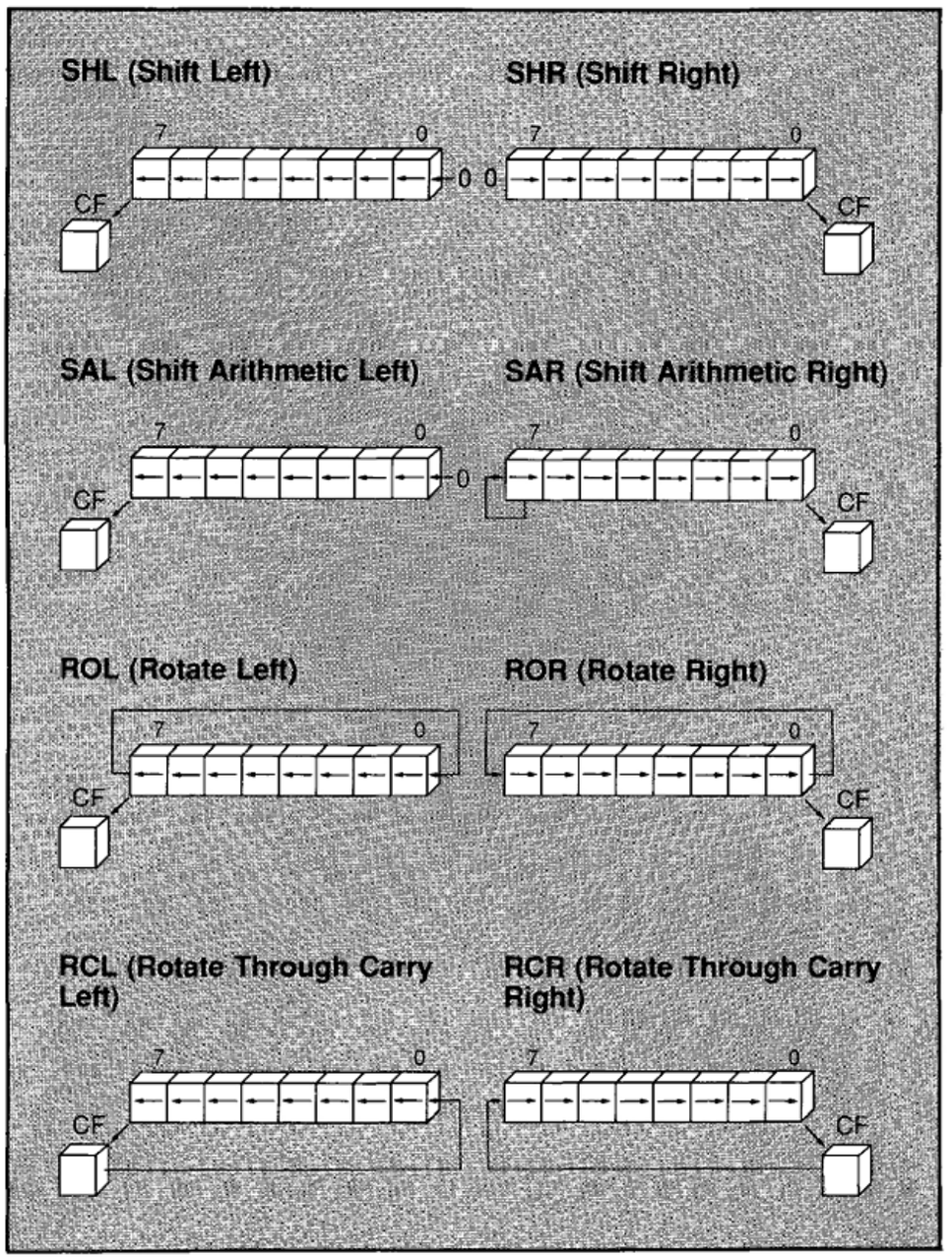


Figure 16.1 Shifts and Rotates

■ **Syntax**

SHL {*register* | *memory*},{**CL** | **1**}
SHR {*register* | *memory*},{**CL** | **1**}
SAL {*register* | *memory*},{**CL** | **1**}
SAR {*register* | *memory*},{**CL** | **1**}
ROL {*register* | *memory*},{**CL** | **1**}
ROR {*register* | *memory*},{**CL** | **1**}
RCL {*register* | *memory*},{**CL** | **1**}
RCR {*register* | *memory*},{**CL** | **1**}

The format of all the shift instructions is the same. The destination operand should contain the value to be shifted. It will contain the shifted operand after the instruction. The source operand should contain the number of bits to shift or rotate. It can be the immediate value 1 or the **CL** register. No other value or register is accepted on the 8088 and 8086 processors.

80186/286/386 Only

Starting with the 80186 processor, 8-bit immediate values larger than 1 can be given as the source operand for shift or rotate instructions, as shown below:

```
shr     bx,4        ; 9 clocks, 3 bytes on 80286
```

The following statements are equivalent if the program must run the 8088 or 8086:

```
mov     cl,4        ; 2 clocks, 3 bytes on 80286
shr     bx,cl       ; 9 clocks, 2 bytes on 80286
                    ;11 clocks, 5 bytes
```

16.8.1 Multiplying and Dividing by Constants

Shifting right by one has the effect of dividing by two; shifting left by one has the effect of multiplying by two. You can take advantage of this to do fast multiplication and division by common constants. The easiest constants are the powers of two. Shifting left twice multiplies by four, shifting left three times multiplies by eight, and so on.

SHR is used to divide unsigned numbers. **SAR** can be used to divide signed numbers, but **SAR** rounds negative numbers down—**IDIV** always rounds up. Code that divides by using **SAR** must adjust for this difference. Multiplication by shifting is the same for signed and unsigned

numbers, so either **SAL** or **SHL** can be used. Both instructions do the same operation.

Since the multiply and divide instructions are the slowest on the 8088 and 8086 processors, using shifts instead can often speed operations by a factor of 10 or more. For example, on the 8088 or 8086 processor, the following statements take 4 clocks:

```
xor     ah,ah       ; Clear AH
shl     ax,1        ; Multiply byte in AL by 2
```

The following statements have the same effect, but take between 74 and 81 clocks on the 8088 or 8086:

```
mov     bl,2        ; Multiply byte in AL by 2
mul     bl
```

The same statements take 15 clocks on the 80286 or between 11 and 16 clocks on the 80386. See the *Microsoft Macro Assembler Reference* for complete information on timing of instructions.

Shift instructions can be combined with add or subtract instructions to do multiplication by common constants. These operations are best put in macros so that they can be changed if the constants in a program change.

■ Example 1

```
mul_10      MACRO   factor      ; Factor must be unsigned
            mov     ax,factor   ; Load into AX
            shl     ax,1        ; AX = factor * 2
            mov     bx,ax       ; Save copy in BX
            shl     ax,1        ; AX = factor * 4
            shl     ax,1        ; AX = factor * 8
            add     ax,bx       ; AX = (factor * 8) + (factor * 2)
            ENDM                ; AX = factor * 10
```

■ Example 2

```
div_u512    MACRO   dividend    ; Dividend must be unsigned
            mov     ax,dividend; Load into AX
            shr     ax,1        ;   AX = dividend / 2 (unsigned)
            xchg    al,ah       ; xchg is like rotate right 8
                                ;   AL = (dividend / 2) / 256
            cbw                 ; Clear upper byte
            ENDM                ;   AX = (dividend / 512
```

16.8.2 Moving Bits to the Least-Significant Position

Sometimes a group of bits within an operand needs to be treated as a single unit—for example, to do an arithmetic operation on those bits without affecting other bits. This can be done by masking off the bits, and then shifting them into the least-significant positions. After the arithmetic operation is done, the bits are shifted back to the original position and merged with the original bits by using **OR**. See Section 17.2.5.2, "Defining and Redefining Interrupt Routines," for an example of this operation.

16.8.3 Adjusting Masks

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position.

■ Example

```
            .DATA
masker      DB      00000010b   ; Mask that may change at run time
            .CODE
            .
            .
            .
            mov     cl,2        ; Rotate two at a time
            mov     bl,57h      ; Load value to be changed    01010111b
            rol     masker,cl   ; Rotate two to left          00001000b
            or      bl,masker   ; Turn on masked values       ---------
                                ; New value is 05Fh           01011111b
            rol     masker,cl   ; Rotate two more             00100000b
            or      bl,masker   ; Turn on masked values       ---------
                                ; New value is 07Fh           01111111b
```

This technique is useful only if the mask value is unknown until runtime.

16.8.4 Shifting Multiword Values

Sometimes it is necessary to shift a value that is too large to fit in a register. In this case, you can shift each part separately, passing the shifted bits through the carry flag. The **RCR** or **RCL** instructions must be used to move the carry value from the first register to the second.

RCR and **RCL** can also be used to initialize the high or low bit of an operand. Since the carry flag is treated as part of the operand (like using a 9-bit operand), the flag value before the operation is crucial. The carry flag may be set by a previous instruction, or you can set it directly using the **CLC** (Clear Carry Flag), **CMC** (Complement Carry Flag), and **STC** (Set Carry Flag) instructions.

■ Example

```
            .DATA
mem32       DD      500000
            .CODE
            .
            .                                   ; Divide 32-bit unsigned by 16
            .
            mov     cx,4                        ; Shift right 4         500000
again:      shr     WORD PTR mem32[2],1         ; Shift into carry  DIV     16
            rcr     WORD PTR mem32[0],1         ; Rotate carry in       ------
            loop    again                       ;                        31250
```

16.8.5 Shifting Multiple Bits

■ 80386 Only

The 80836 processor has new instructions for shifting multiple bits into an operand. The **SHLD** (Double Precision Shift Left) instruction shifts a specified group of bits left and into an operand. The **SHRD** (Double Precision Shift Right) instruction shifts a specified group of bits right and into an operand.

■ Syntax

SHRD {*register* | *memory*},*register*,{**CL** | *immediate*}
SHLD {*register* | *memory*},*register*,{**CL** | *immediate*}

These instructions take three operands. The first (leftmost) contains the value to be shifted. It must be a 16-bit or 32-bit register or memory operand. The second operand contains the bits to be shifted into the value. It must be a register of the same size as the first operand. The third operand contains the number of bits to shift. It may be an immediate operand or the **CL** register.

■ Example

```
            mov     ax,3AF2h ; Load    AX=00111010 11110010
            mov     bx,9C00h ; Load    BX=                  10011100 00000000
            shld    ax,bx,7  ; Shift 7    01111001 0        <- 7
                             ;                 1001110 <- 7
                             ;              ----------------
                             ;           AX=01111001 01001110 (794Eh)
```

CHAPTER 17

CONTROLLING PROGRAM FLOW

17.1 Jumping 333
17.1.1 Jumping Unconditionally 333
17.1.2 Jumping Conditionally 335
17.1.2.1 Comparing and Jumping 335
17.1.2.2 Jumping Based on Flag Status 338
17.1.2.3 Testing Bits and Jumping 340
17.1.2.4 Testing and Setting Bits 341
17.2 Looping 343
17.3 Setting Bytes Conditionally 345
17.4 Using Procedures 346
17.4.1 Calling Procedures 347
17.4.2 Defining Procedures 347
17.4.3 Passing Arguments on the Stack 349
17.4.4 Using Local Variables 351
17.4.5 Setting Up Stack Frames 354
17.5 Using Interrupts 355
17.5.1 Calling Interrupts 356
17.5.2 Defining and Redefining Interrupt Routines 358
17.6 Checking Memory Ranges 361

The 8086-family processors provide a variety of instructions for controlling the flow of a program. The four major types of program-flow instructions are jumps, loops, procedure calls, and interrupts.

This chapter tells you how to use these instructions and how to test conditions for the instructions that change program flow conditionally.

17.1 Jumping

Jumps are the most direct method of changing program control from one location to another. At the internal level, jumps work by changing the value of the IP (Instruction Pointer) register from the address of the current instruction to a target address.

Jumps can be short, near, or far. **MASM** automatically handles near and short jumps, though it may not always generate the most efficient code if the label being jumped to is a forward reference. The size and control of jumps is discussed in Section 9.4.1, "Forward References to Labels."

17.1.1 Jumping Unconditionally

The **JMP** instruction is used to jump unconditionally to a specified address.

■ Syntax

JMP {*register* | *memory*}

The operand should contain the address to be jumped to. Unlike conditional jumps, whose target address must be short (within 128 bytes), the target address for unconditional jumps can be short, near, or far. See Section 9.4.1, "Forward References to Labels," for more information on specifying the distance for conditional jumps.

If a conditional jump must be greater than 128 bytes, the construction must be reorganized (except on the 80386). This can be done by reversing the sense of the conditional jump and adding an unconditional jump, as shown in Example 1.

■ Example 1

```
            cmp     ax,7        ; If AX is 7 and jump is short
            je      close       ;   then jump close

            cmp     ax,6        ; If AX is 6 and jump is near
            jne     close       ;   then test opposite and skip over
            jmp     distant     ; Now jump
            .
            .
            .
close:                          ; Less than 128 bytes from jump
            .
            .
            .
distant:                        ; More than 128 bytes from jump
```

An unconditional jump can be used as a form of conditional jump by specifying the address in a register or indirect memory operand. The value of the operand can be calculated at run time, based on user interaction or other factors. You can use indirect memory operands to construct jump tables that work like C **switch** statements, BASIC **ON GOTO** statements, or Pascal **case** statements.

■ Example 2

```
            .CODE
            .
            .
            .
            jmp     process             ; Jump over data
ctl_tbl     LABEL   WORD                ;   (required in overlay procedures)
            DW      extended            ; Null key (extended code)
            DW      ctrla               ; Address of CONTROL-A key routine
            DW      ctrlb               ; Address of CONTROL-B key routine
process:    mov     ah,8h               ; Get a key
            int     21h
            cbw                         ; Convert AL to AX
            mov     bx,ax               ; Copy
            shl     bx,1                ; Convert to address

            jmp     ctl_tbl[bx]         ; Jump to key routine

extended:   mov     ah,8h               ; Get second key of extended
            int     21h
            .                           ; Use another jump table
            .                           ;   for extended keys
            .
ctrla:      .                           ; CONTROL-A routine here
            .
            .
            jmp     next

ctrlb:      .                           ; CONTROL-B routine here
            .
            .
            jmp     next
            .
            .
next:       .                           ; Continue
```

In Example 2, an indirect memory operand points to addresses of routines for handling different keystrokes. Notice that the jump table is placed in the code segment. This technique is optional in stand-alone assembler programs, but it may be required for procedures called from some languages.

17.1.2 Jumping Conditionally

The most common way of transferring control in assembly language is with conditional jumps. This is a two-step process: first test the condition, and then jump if the condition is true or continue if it is false.

■ Syntax

***J**condition label*

Conditional-jump instructions take a single operand containing the address to be jumped to. The distance from the jump instruction to the specified address must be short (less than 128 bytes). If a longer distance is specified, an error will be generated telling the distance of the jump in bytes. See Section 17.1.1, "Jumping Unconditionally," for information on arranging longer conditional jumps.

80386 Only

Conditional jumps to forward references are near by default under the 80386 processor. But you can use the **SHORT** operator to specify short jumps. See Section 9.4.1, "Forward References to Labels," for information specifying the size of jumps.

Conditional-jump instructions (except **JCXZ**) use the status of one or more flags as their condition. Thus any statement that sets a flag under specified conditions can be the test statement. The most common test statements use the **CMP** or **TEST** instructions. The jump statement can be any one of 31 conditional-jump instructions.

17.1.2.1 Comparing and Jumping

The **CMP** instruction is specifically designed to test for conditional jumps. It does not change the destination operand, so it can be used to compare two values without changing either of them. Instructions that change operands (such as **SUB** or **AND**) can also be used to test conditions.

The **CMP** instruction compares two operands and sets flags based on the result. It is used to test the following relationships: equal; not equal; greater than; less than; greater than or equal; or less than or equal.

■ Syntax

CMP {*register* | *memory*},{*register* | *memory* | *immediate*}

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, they cannot both be memory operands.

The jump instructions that can be used with **CMP** are made up of mnemonic letters combined to indicate the type of jump. The letters are shown below:

Letter	Meaning
J	Jump
G	Greater than (for unsigned comparisons)
L	Less than (for unsigned comparisons)
A	Above (for signed comparisons)
B	Below (for signed comparisons)
E	Equal
N	Not

The mnemonic names always refer to the relationship that the first operand of the **CMP** instruction has to the second operand of the **CMP** instruction. For instance, **JG** tests whether the first operand is greater than the second. Several conditional instructions have two names. You can use whichever name seems more mnemonic in context.

Comparisons and conditional jumps can be thought of as statements in the following format:

IF (*value1 relationship value2*) **THEN GOTO** *truelabel*

Statements of this type can be coded in assembly language by using the following syntax:

CMP *value1,value2*
J*relationship truelabel*
.
.
.
truelabel:

Table 17.1 lists conditional-jump instructions for each *relationship* and shows the flags that are tested in order to see if *relationship* is true.

Table 17.1

Conditional-Jump Instructions Used after Compare

Jump Condition		Signed Compare	Jump if:	Unsigned Compare	Jump if:
Equal	=	**JE**	**ZF**=1	**JE**	**ZF**=1
Not equal	≠	**JNE**	**ZF**=1	**JNE**	**ZF**=1
Greater than	>	**JG** or **JNLE**	**ZF**=0 and **SF**=**OF**	**JA** or **JNBE**	**CF**=0 and **ZF**=0
Less than or equal	≤	**JLE** or **JNG**	**ZF**=1 and **SF**≠**OF**	**JBE** or **JNA**	**CF**=1 or **ZF**=1
Less than	<	**JL** or **JNGE**	**SF**≠**OF**	**JB** or **JNAE**	**CF**=1
Greater than or equal	≥	**JGE** or **JNL**	**SF**=**OF**	**JAE** or **JNB**	**CF**=0

Internally, the **CMP** instruction is exactly the same as the **SUB** instruction, except that the destination operand is not changed. The flags are set according to the result that would have been generated by a subtraction.

■ Example 1

```
; If CX is less than -20, then make DX 30, else make DX 20

            cmp     cx,-20      ; If signed CX is smaller than -20
            jl      less        ;    Then do stuff at "less"
            mov     dx,20       ; Else set DX to 20
            jmp     further     ; Finished
less:       mov     dx,30       ; Then set DX to 30
further:
```

Example 1 shows the basic form of conditional jumps. Notice that in assembly language, if-then-else constructions are usually written in the form if-else-then.

This theme has many variations. For example, you may find it more mnemonic to code in the if-then-else format. However, you must then use the opposite jump condition, as shown in Example 2.

■ Example 2

```
; If CX is greater than or equal to -20, then make DX 20, else make DX 30

            cmp     cx,-20      ; If signed CX is smaller than -20
            jnl     notless     ;    else do stuff at "notless"
            mov     dx,30       ; Then set DX to 30
            jmp     continue    ; Finished
notless:    mov     dx,20       ; Else set DX to 20
continue:
```

The then-if-else format shown in Example 3 is often more efficient. Do the work for the most likely case, and then compare for the opposite condition. If the condition is true, you are finished.

■ Example 3

```
; DX is 20, unless CX is less than -20, then make DX 30

            mov     dx,20       ; DX is 20
            cmp     cx,-20      ; If signed CX is greater than -20
            jge     greatequ    ;    Then done
            mov     dx,30       ; Else set DX to 30
greatequ:
```

This example avoids the unconditional jump used in Examples 1 and 2 and thus is faster even if the less likely condition is true.

17.1.2.2 Jumping Based on Flag Status

The **CMP** instruction is the most mnemonic way to set the flags for conditional jumps, but any instruction that changes flags can be used as the test condition. The conditional-jump instructions listed below enable you to jump based on the condition of flags rather than on relationships of operands. Some of these instructions have the same effect as instructions listed in Table 17.1.

Instruction	Action
JO	Jumps if the overflow flag is set
JNO	Jumps if the overflow flag is clear
JC	Jumps if the carry flag is set (same as **JB**)
JNC	Jumps if the carry flag is clear (same as **JAE**)
JZ	Jumps if the zero flag is set (same as **JE**)
JNZ	Jumps if the zero flag is clear (same as **JNE**)
JS	Jumps if the sign flag is set
JNS	Jumps if the sign flag is clear
JP	Jumps if the parity flag is set
JNP	Jumps if the parity flag is clear
JPE	Jumps if parity is even (parity flag set)
JPO	Jumps if parity is odd (parity flag clear)
JCXZ	Jumps if **CX** is 0

Notice that the **JCXZ** is the only conditional jump based on the condition of a register (**CX**) rather than flags. Since **JCXZ** is usually used with loop instructions, it is discussed in more detail in Section 17.3, "Setting Bytes Conditionally."

■ Example 1

```
            add     ax,bx       ; Add two values
            jo      overflow    ; If value too large, adjust
            .
            .
            .
overflow:                       ; Adjustment routine here
```

■ Example 2

```
            sub     ax,dx       ; Subtract
            jnz     go_on       ; If the result is not zero, continue
            call    zhandler    ;    else do special case
go_on:
```

17.1.2.3 Testing Bits and Jumping

Like the **CMP** instruction, the **TEST** instruction is designed to test for conditional jumps. However, specific bits are compared rather than entire operands.

■ Syntax

TEST {*register* | *memory*},{*register* | *memory* | *immediate*}

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, the operands cannot both be memory.

Normally, one of the operands is a mask in which the bits to be tested are the only bits set. The other operand contains the value to be tested. If all the bits set in the mask are clear in the operand being tested, the zero flag will be set. If any of the flags set in the mask are also set in the operand, the zero flag will be cleared.

The **TEST** instruction is actually the same as the **AND** instruction, except that neither operand is changed. If the result of the operation is 0, the zero flag is set, but the 0 is not actually written to the destination operand.

You can use the **JZ** and **JNZ** instructions to jump after the test. **JE** and **JNE** are the same and can be used if you find them more mnemonic.

■ Example

```
            .DATA
bits        DB      ?
            .CODE
            .
            .
            .
; If bit 2 or bit 4 is set, then call taska

                                ; Assume "bits" is 0D3h       11010011
            test    bits,10100b; If 2 or 4 is set        AND 00010100
            jz      go_on       ; Else continue              --------
            call    taska       ;    Then call taska         00010000
go_on:                          ; Jump not taken
            .
            .
            .
; If bits 2 and 4 are clear, then call taskb

                                ; Assume "bits" is 0E9h       11101001
            test    bits,10100b; If 2 and 4 are clear    AND 00010100
            jnz     next        ; Else continue              --------
            call    taskb       ;    Then call taskb         00000000
next:                           ; Jump not taken
```

17.1.2.4 Testing and Setting Bits

■ **80386 Only**

The 80386 processor has bit test and set instructions. These instructions have two purposes. They can test the status of a bit to control program flow; some of them can also change the value of a specified bit.

■ **Syntax**

BT {*register* | *memory*},{*register* | *immediate*}
BTC {*register* | *memory*},{*register* | *immediate*}
BTR {*register* | *memory*},{*register* | *immediate*}
BTS {*register* | *memory*},{*register* | *immediate*}

For each of the instructions, the memory or register destination operand is the target value that will be tested. The register or immediate source operand specifies the number of the bit to be tested in the destination operand. The four bit-testing instructions are described below:

Instruction	Description
BT	The Bit Test instruction examines the specified bit in the target value and puts a copy in the carry flag. The carry flag can then be used by another instruction such as a conditional jump. For example, assume **BX** points to a bit field and **CX** contains 4 in the following statements:

```
bt      [bx],cx       ; Put bit 4 of bit field
                      ;   pointed to by BX in carry
jc      somewhere     ; Jump if carry set
```

The same thing could be done less efficiently on other 8086-family processors with the following statements:

```
mov     ax,[bx]     ; Load value pointed to by BX
shr     ax,cl       ; Shift bit 4 to first position
test    ax,1        ; See if bit is set
jnz     somewhere   ; Jump if it is
```

This instruction is only useful if the source operand is not known until run time. If the source operand is a constant, the **TEST** instruction (see Section 17.1.2.3, "Testing Bits and Jumping") is more efficient.

BTC The Bit Test and Complement instruction examines the specified bit in the target value and puts a copy in the carry flag. It then reverses the value of the bit. For example, assume **BX** points to a bit field and **CX** contains 4 in the following statements:

```
btc     [bx],cx     ; Put bit 4 of bit field in carry
                    ;   and toggle bit 4
jc      somewhere   ; Jump if carry set
```

BTR The Bit Test and Reset instruction examines the specified bit in the target value and puts a copy in the carry flag. It then clears the bit. For example, assume **BX** points to a bit field and **CX** contains 4 in the following statements:

```
btr     [bx],cx     ; Put bit 4 of bit field in carry
                    ;   and clear bit 4
jc      somewhere   ; Jump if carry set
```

BTS The Bit Test and Set instruction examines the specified bit in the target value and puts a copy in the carry flag. It then sets the bit. For example, assume **BX** points to a bit field and **CX** contains 4 in the following statements:

```
bts     [bx],cx     ; Put bit 4 of bit field in carry
                    ;   and set bit 4
jc      somewhere   ; Jump if carry was set
```

■ Example

```
            .DATA
flag        RECORD  a:3=0,b:2=0,c:1=0,d:2=0,e:1=0,f:1=0
error       flag    <>
            .CODE
            .
            .
            .
            btr     error,c
            jc      fixc
            .
            .
fixa:       .
```

In this example, a bit field made up of error flags is tested. If the bit flag being tested is set, indicating an error, the flag is turned off and control is directed to a label where the error is corrected.

17.2 Looping

The 8086-family of processors has several instructions specifically designed for creating loops of repeated instructions. In addition, you can create loops using conditional jumps.

■ Syntax

LOOP *label*
LOOPE *label*
LOOPZ *label*
LOOPNE *label*
LOOPNZ *label*
JCXZ *label*

The **LOOP** instruction is used for loops with a set number of iterations. For example, it can be used in constructions similar to the "for" loops of BASIC, C, and Pascal, and the "do" loops of FORTRAN.

A single operand specifies the address to jump to each time through the loop. The **CX** register is used as a counter for the number of times to loop. On each iteration, **CX** is decremented. When **CX** reaches 0, control passes to the instruction after the loop.

The **LOOPE**, **LOOPZ**, **LOOPNE**, and **LOOPNZ** instructions are used in loops that check for a condition. For example, they can be used in constructions similar to the "while" loops of BASIC, C, and Pascal; the "repeat" loops of Pascal; and the "do" loops of C.

The **LOOPE** (also called **LOOPZ**) instruction can be thought of as meaning "loop while equal." Similarly, **LOOPNE** (also called **LOOPNZ**) instruction can be thought of as meaning "loop while not equal." A single short memory operand specifies the address to loop to each time through. The **CX** register can specify a maximum number of times to go through the loop. The **CX** register can be set to a number that is out of range if you do not want a maximum count.

The **JCXZ** instruction (and its 32-bit 80386 extension, **JECXZ**) are often used in loop structures. For example, it may be used in loops that check a condition at the start of the loop rather than at the end. Unlike the loop instruction, **JCXZ** does not decrement **CX**, so the programmer must use another statement to decrement the count.

■ 80386 Only

Unlike conditional-jump instructions, which can jump to either a near or a short label under the 80386, the loop instructions, **JCXZ** instruction, and **JECXZ** instruction always jump to a short label.

■ Example 1

```
; For 0 to 200 do task

            mov     cx,200              ; Set counter
next:       .                           ; Do the task here
            .
            .
            loop    next                ; Do again
                                        ; Continue after loop
```

This loop has the same effect as the following statements:

```
; For 0 to 200, do task

            mov     cx,200              ; Set counter
next:       .
            .                           ; Do the task here
            .
            dec     cx
            cmp     cx,0
            jne     next                ; Do again
                                        ; Continue after loop
```

The first version is more efficient as well as easier to understand. However, there are situations in which you must use conditional-jump instructions rather than loop instructions. For example, conditional jumps are often required for loops that test several conditions.

If the counter in **CX** is variable because of previous instructions, you should use the **JCXZ** instruction to check for 0, as shown in Example 2. Otherwise, if **CX** is 0, it will be decremented to –1 in the first iteration and will continue through 65,535 iterations before it reaches 0 again.

■ Example 2

```
; For 0 to CX do task

                                        ; CX counter set previously
            jcxz    done                ; Check for 0
next:       .                           ; Do the task here
            .
            .
            loop    next                ; Do again
done:                                   ; Continue after loop
```

■ Example 3

```
; While AX is not 128, do task

            mov     cx,OFFFFh           ; Set count too high to interfere
wend:       .                           ; Do the task here
            .
            .
            cmp     ax,128              ; Is it 128?
            loopne  wend                ; No? Repeat
                                        ; Yes? Continue
```

17.3 Setting Bytes Conditionally

■ 80386 Only

The 80386 processor has a new group of instructions for setting bytes conditionally. These instructions test the condition of specified flags, and depending on the result, set a memory operand either to 1 or to 0. They can be used to set byte variables that are used as Boolean flags.

■ Syntax

SET*condition* {*register* | *memory*}

Conditional-set instructions test conditions in the same way as conditional-jump instructions, except that instead of jumping if the condition is met, they set a specified byte. For example, **SETZ** is similar to **JZ**, **SETNE** is similar to **JNE**, and so on. See Section 17.1.1, "Jumping Unconditionally," for more information on how flags are tested for conditional jumps.

Conditional-set instructions require one 8-bit operand, which can be either a register or a memory operand. If the condition tested by the instruction is true, the operand is set to 1. Otherwise the operand is set to 0.

Conditional-set instructions are usually preceded by a **CMP** or **TEST** instruction, although any instruction that sets flags can be used to test for the condition.

■ Example

```
            .DATA
bigflag     DB      ?           ; Boolean flag
amount      DW      ?           ; Size variable to be set at run time
            .CODE
            .
            .                   ; Size is set
            .
; bigflag = amount > 1000

            cmp     size,1000   ; Is "size" greater than 1000?
            setg    bigflag     ; If greater, "bigflag" = 1
                                ;   else "bigflag" = 0
```

In the example, the Boolean variable `bigflag` is set according to a comparison of two other values. Some languages (such as BASIC) set the result of true relational statements to –1 rather than 1. To make the code compatible with such compilers, you should negate the value after setting it. For example, add the following line to the previous example:

```
            neg     bigflag     ; Negate result
```

This statement would be necessary for BASIC, since the expression `BIGFLAG=SIZE>1000` evaluates to –1. It would not be necessary for C, since the expression `bigflag=size>1000` evaluates to 1.

17.4 Using Procedures

Procedures are units of code that do a specific task. They provide a way of modularizing code so that a task can be accomplished from any point in a program without using the same code in each place. Assembly-language procedures are comparable to functions in C; subprograms, functions, and subroutines in BASIC; procedures and functions in Pascal; or routines and functions in FORTRAN.

Two instructions and two directives are usually used in combination to define and use assembly-language procedures. The **CALL** instruction is used to call procedures defined elsewhere. The **RET** instruction is used to return control from a called procedure to the code that called it. The **PROC** and **ENDP** directives normally mark the beginning and end of a procedure definition, as described in Section 17.4.2, "Defining Procedures."

The **CALL** and **RET** instructions use the stack to keep track of the location of the procedure. The **CALL** instruction pushes the calling address onto the stack and then jumps to the starting address of the procedure.

The **RET** instruction pops the address pushed by the **CALL** instruction and returns control to the instruction following the call.

Every **CALL** must have a **RET** to restore the stack to its status before the **CALL**. Calls may be nested.

17.4.1 Calling Procedures

The **CALL** instruction saves the address following the instruction on the stack and passes control to a specified address.

■ Syntax

CALL {*register* | *memory*}

The address is usually specified as a direct memory operand. However, the operand can also be a register or indirect memory operand containing a value calculated at run time. This enables you to write call tables similar to the jump table illustrated in Section 17.1.2.1, "Comparing and Jumping."

Calls can be near or far. Near calls push only the offset portion of the calling address. Far calls push both the segment and offset. You must give the type of far calls to forward-referenced labels using the **FAR** type specifier and the **PTR** operator. For example, use the following statement to make a far call to a label that has not been earlier defined or declared external in the source code:

```
call    FAR PTR task
```

17.4.2 Defining Procedures

Procedures are defined by labeling the start of the procedure and placing a **RET** instruction at the end. There are several variations on this syntax.

■ Syntax 1

label **PROC** [[**NEAR** | **FAR**]]
statements
RET [[*constant*]]
label **ENDP**

Procedures are normally defined by using the **PROC** directive at the start of the procedure and the **ENDP** directive at the end. The **RET** instruction is normally placed immediately before the **ENDP** directive. The size of the **RET** instruction automatically matches the size defined by the **PROC** directive.

■ Syntax 2

label:
statements
RETN [[*constant*]]

■ Syntax 3

label **LABEL FAR**
statements
RETF [[*constant*]]

Starting with Version 5.0 of the Macro Assembler, the **RET** instruction can be extended to **RETN** (Return Near) to override the default size. This enables you to define and use procedures without the **PROC** and **ENDP** directives, as shown in Syntax 2 and Syntax 3 above. However, with this method, the programmer is responsible for making sure the size of the **CALL** matches the size of the **RET**.

The **RET** instruction (and its **RETF** and **RETN** variations) allows a constant operand that specifies a number of bytes to be added to the value of the **SP** register after the return. This operand can be used to adjust for arguments passed to the procedure before the call, as shown in the example in Section 17.4.4, "Using Local Variables."

■ Example 1

```
            call    task            ; Call is near because procedure is near
            .                       ; Return comes to here
            .
            .
task        PROC    NEAR            ; Define "task" to be near
            .
            .                       ; Instructions of "task" go here
            .
            ret                     ; Return to instruction after call
task        ENDP                    ; End "task" definition
```

Example 1 shows the recommended way of making calls with **MASM**. Example 2 shows another method that programmers who are used to other assemblers may find more familiar.

■ **Example 2**

```
            call    NEAR PTR task ; Call is declared near
            .                     ; Return comes to here
            .
            .
task:                             ; Procedure begins with near label
            .
            .                     ; Instructions go here
            .
            retn                  ; Return declared near
```

This method gives more direct control over procedures, but the programmer must make sure that calls have the same size as corresponding returns.

For example, if a call is made with the statement

```
call NEAR PTR task
```

the assembler does a near call. This means that one word (the offset following the calling address) is pushed onto the stack. If the return is made with the statement

```
retf
```

two words are popped off the stack. The first will be the offset, but the second will be whatever happened to be on the stack before the call. Not only will the popped value be meaningless, but the stack status will be incorrect, causing the program to fail.

17.4.3 Passing Arguments on the Stack

Procedure arguments can be passed in various ways. For example, values can be passed to a procedure in registers or in variables. However, the most common method of passing arguments is to use the stack. Microsoft languages have a specific convention for doing this.

The arguments are pushed onto the stack before the call. After the call, the procedure retrieves and processes them. At the end of the procedure, the stack is adjusted to account for the arguments.

Although the same basic method is used for all Microsoft high-level languages, the details vary. For instance, in some languages, pointers to the arguments are passed to the procedure; in others the arguments themselves are passed. The order in which arguments are passed (whether the first argument is pushed first or last) also varies according the language. Finally, in some languages, the stack is adjusted by the **RET** instruction in the called procedure; in others the code immediately following the **CALL** instruction adjusts the stack. See the *Microsoft Mixed-Language Programming Guide* for details on calling conventions for each Microsoft language.

■ Example

```
; C-style procedure call and definition

            mov     ax,10       ; Load and
            push    ax          ;   push constant as third argument
            push    arg2        ; Push memory as second argument
            push    cx          ; Push register as first argument
            call    addup       ; Call the procedure
            add     sp,6        ; Destroy the pushed arguments
            .                   ;   (equivalent to three pops)
            .
            .
addup       PROC    NEAR        ; Return address for near call
                                ;   takes two bytes
            push    bp          ; Save base pointer - takes two bytes
                                ;   so arguments start at 4th byte
            mov     bp,sp       ; Load stack into base pointer
            mov     ax,[bp+4]   ; Get first argument from
                                ;   4th byte above pointer
            add     ax,[bp+6]   ; Add second argument from
                                ;   6th byte above pointer
            add     ax,[bp+8]   ; Add third argument from
                                ;   8th byte above pointer
            pop     bp          ; Restore BP
            ret                 ; Return result in AX
addup       ENDP
```

The example shows one method of passing arguments to a procedure. This method is similar to the way procedures are called in C. Figure 17.1 shows the stack condition at key points in the process.

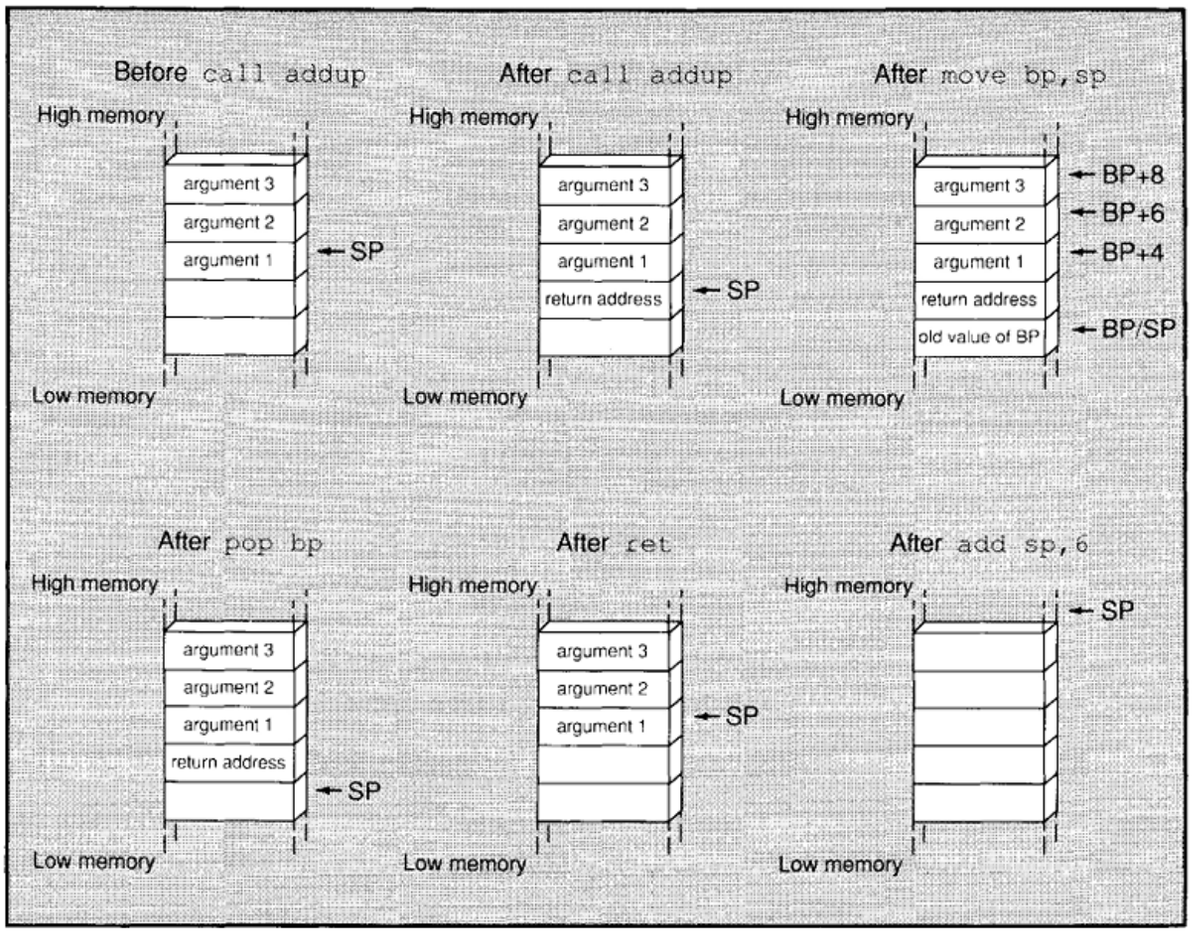


Figure 17.1 Procedure Arguments on the Stack

Note

Arguments passed on the stack in assembler routines cannot be accessed by name with the CodeView debugger. They can be accessed by an expression that specifies their stack position.

17.4.4 Using Local Variables

In high-level languages, local variables are variables known only within a procedure. In Microsoft languages, these variables are usually stored on

the stack. Assembly-language programs can use the same concept. These variables should not be confused with labels or variable names that are local to a module, as described in Chapter 8, "Creating Programs from Multiple Modules."

Local variables are created by saving stack space for the variable at the start of the procedure. The variable can then be accessed by its position in the stack. At the end of the procedure, the stack pointer is restored to restore the memory used by local variables.

■ Example

```
            push    ax          ; Push one argument
            call    task        ; Call
            .
            .
            .
arg         EQU     <[bp+4]>    ; Name for argument
loc         EQU     <[bp-2]>    ; Name for local variable

task        PROC    NEAR
            push    bp          ; Save base pointer
            mov     bp,sp       ; Load stack into base pointer
            sub     sp,2        ; Save two bytes for local variable
            .
            .
            .
            mov     loc,3       ; Initialize local variable
            add     ax,loc      ; Add local variable to AX
            sub     arg,ax      ; Subtract local from argument
            .                   ; Use "loc" and "arg" in other operations
            .
            .
            mov     sp,bp       ; Adjust for stack variable
            pop     bp          ; Restore base
            ret     2           ; Return result in AX and pop
task        ENDP                ;   two bytes to adjust stack
```

In this example, two bytes are subtracted from the **SP** register to make room for a local word variable. This variable can then be accessed as `[bp-2]`. In the example, this value is given the name `loc` with a text equate. Notice that the instruction `mov sp,bp` is given at the end to restore the original value of **SP**. The statement is only required if the value of **SP** is changed inside the procedure (usually by allocating local variables).

The argument passed to the procedure is returned with the **RET** instruction. Contrast this to the example in Section 17.4.3, "Passing Arguments on the Stack," in which the calling code adjusts for the argument. Figure 17.2 shows the state of the stack at key points in the process.

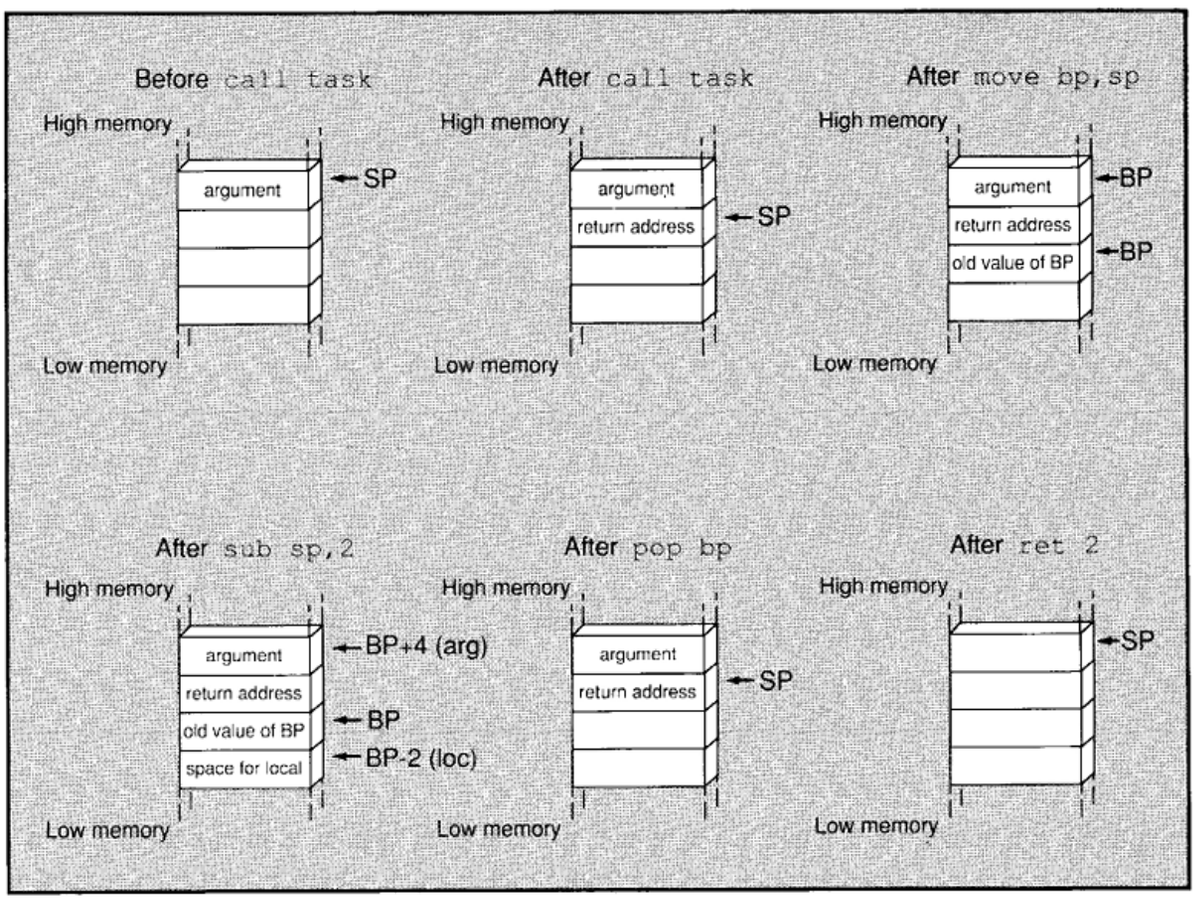


Figure 17.2 Local Variables on the Stack

Note

Local variables created in assembler routines cannot be accessed by name with the CodeView debugger. They can be accessed by an expression that specifies their stack position.

17.4.5 Setting Up Stack Frames

■ 80186/286/386 Only

Starting with the 80186 processor, the **ENTER** and **LEAVE** instructions are provided for setting up a stack frame. These instructions do the same thing as the multiple instructions at the start and end of procedures in the Microsoft calling conventions (see the examples in Section 17.4.3, "Passing Arguments on the Stack").

■ Syntax

ENTER *framesize,nestinglevel*
statements
LEAVE

The **ENTER** instruction takes two constant operands. The *framesize* (a 16-bit constant) specifies how many bytes to reserve for local variables. The *nestinglevel* (an 8-bit constant) specifies the level at which the procedure is nested. This operand should always be 0 when writing procedures for BASIC, C, and FORTRAN. The *nestinglevel* can be greater than 0 with Pascal and other languages that enable procedures to access the local variables of calling procedures.

The **LEAVE** instruction reverses the effect of the last **ENTER** instruction by restoring **BP** and **SP** to their values before the procedure call.

■ Example 1

```
task        PROC    NEAR
            enter   6,0         ; Set stack frame and reserve 6
            .                   ;   bytes for local variables
            .                   ; Do task here
            .
            leave               ; Restore stack frame
            ret                 ; Return
task        ENDP
```

Example 1 has the same effect as the code in Example 2.

■ **Example 2**

```
task        PROC    NEAR
            push    bp          ; Save base pointer
            mov     bp,sp       ; Load stack into base pointer
            sub     sp,6        ; Reserve 6 bytes for local variables
            .
            .                   ; Do task here
            .
            mov     sp,bp       ; Restore stack pointer
            pop     bp          ; Restore base
            ret                 ; Return
task        ENDP
```

The code in Example 1 takes fewer bytes, but is slightly slower. See the *Microsoft Macro Assembler Reference* for exact comparisons of size and timing.

17.5 Using Interrupts

Interrupts are a special form of routines that are called by number instead of by address. They can be initiated by hardware devices as well as by software. Hardware interrupts are called automatically whenever certain events occur in the hardware.

Interrupts can have any number from 0 to 255. Most of the interrupts with lower numbers are reserved for use by the processor, DOS, or the BIOS.

The programmer can call existing interrupts with the **INT** instruction. Interrupt routines can also be defined or redefined to be called later. For example, an interrupt routine that is called automatically by a hardware device can be redefined so that its action is different.

DOS defines several interrupt handlers. Two that are sometimes used by applications programmers are listed below:

Interrupt	Description
0	Divides overflow. Called automatically when the quotient of a divide operation is too large for the source operand or when a divide by zero is attempted.
4	Overflows. Called by the **INTO** instruction if the overflow flag is set.

Interrupt 21h is the current method of using DOS functions. To call a function, place the function number in **AH**, put arguments in registers as appropriate, then call the interrupt. For complete documentation of DOS functions, see the *Microsoft MS-DOS Programmer's Reference* or one of the many other books on DOS functions.

DOS has several other interrupts, but they should not normally be called. Some (such as 20h and 27h) have been replaced by DOS functions. Others are used internally by DOS.

Note

OS/2, the planned multitasking versions of DOS, will not use interrupt 21h. The Application Program Interface (API) will be used instead. This is the method currently used for Microsoft Windows applications.

The BIOS of most computers that use DOS can also be accessed by interrupts. BIOS interrupts are not documented here, since they vary for different computers. See the technical reference documents for your hardware.

17.5.1 Calling Interrupts

Interrupts are called with the **INT** instruction.

■ Syntax

INT *interruptnumber*
INTO

The **INT** instruction takes an immediate operand with a value between 0 and 255.

When calling DOS and BIOS interrupts, a function number is usually placed in the **AH** register. Other registers may be used to pass arguments to functions. Some interrupts and functions return values in certain registers. Register use varies for each interrupt.

When the instruction is called, the processor takes the following six steps:

1. Looks up the address of the interrupt routine in the interrupt descriptor table. In real mode, this table starts at the lowest point in memory (segment 0, offset 0) and consists of four bytes (two segment and two offset) for each interrupt. Thus the address of an

interrupt routine can be found by multiplying the number of the interrupt by four.

2. Pushes the flags register, the current code segment (**CS**), and the current instruction pointer (**IP**).
3. Clears the trap (**TF**) and interrupt enable (**IF**) flags.
4. Jumps to the address of the interrupt routine, as specified in the interrupt description table.
5. Executes the code of the interrupt routine until it encounters an **IRET** instruction.
6. Pops the instruction pointer, code segment, and flags.

Figure 17.3 shows the status of the stack immediately after the **INT** instruction has been executed.

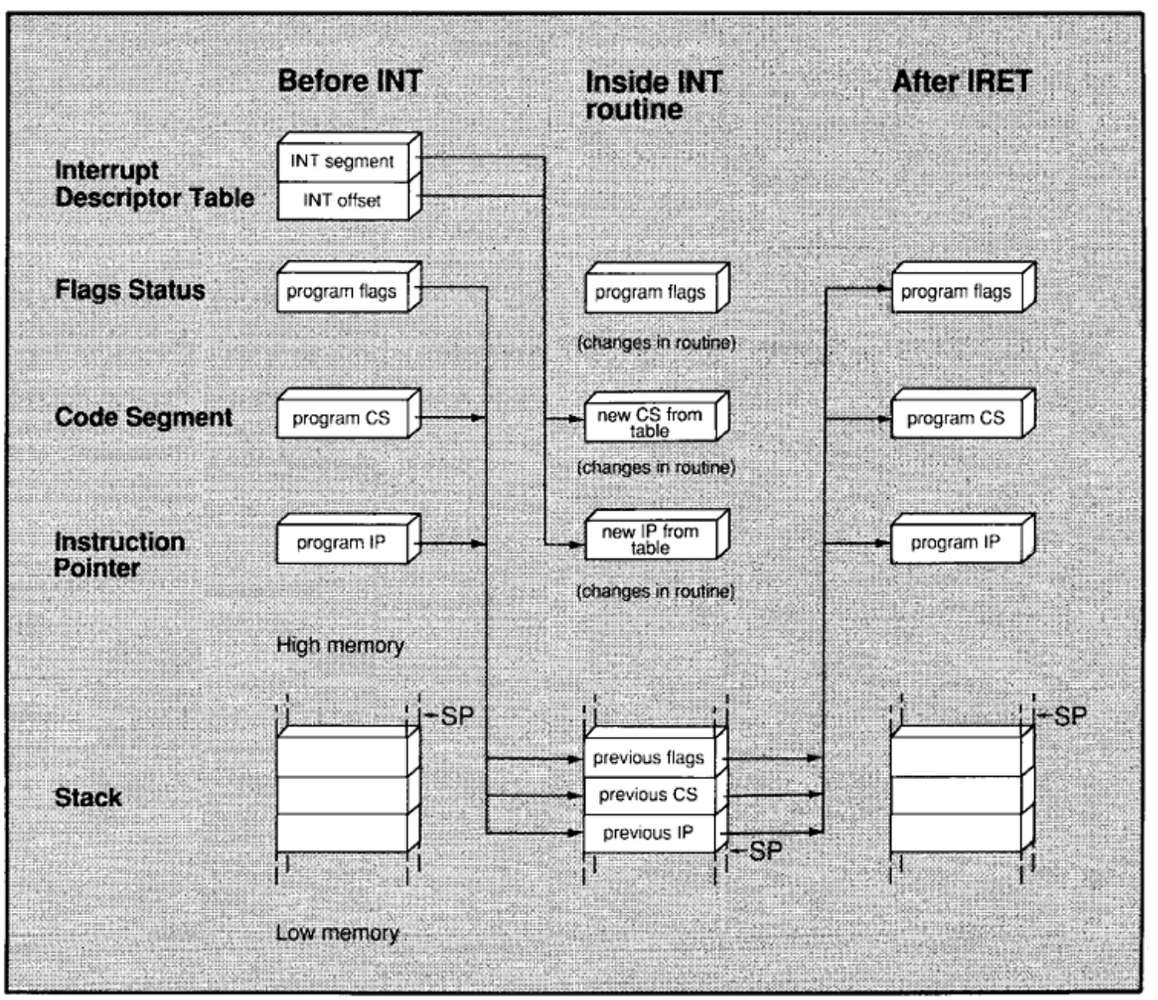


Figure 17.3 Operation of Interrupts

The **INTO** (Interrupt on Overflow) instruction is a variation of the **INT** instruction. It calls interrupt 04h if called when the overflow flag is set. By default, the routine for interrupt 4 simply consists of an **IRET** so that it returns without doing anything. However, you can write your own overflow interrupt routine. Using **INTO** is an alternative to using **JO** (Jump on Overflow) to jump to an overflow routine. Section 17.5.2, "Defining and Redefining Interrupt Routines," gives an example of this.

The **CLI** (Clear Interrupt Flag) and **STI** (Set Interrupt Flag) instructions can be used to turn interrupts on or off. You can use **CLI** to turn interrupt processing off so that an important routine cannot be stopped by a hardware interrupt. After the routine has finished, use **STI** to turn interrupt processing back on. Interrupts received while interrupt processing was turned off by **CLI** are saved and executed when **STI** turns interrupts back on.

■ Example 1

```
; DOS call (Display String)

            mov     ah,09h              ; Load function number
            mov     dx,OFFSET string    ; Load argument
            int     21h                 ; Call DOS
```

■ Example 2

```
; BIOS call (Read Character from Keyboard)

            xor     ah,ah               ; Load function number 0 in AH
            int     16h                 ; Call BIOS
                                        ; Return scan code in AH
                                        ; Return ascii code in AL
```

Example 2 is a BIOS call that works on IBM Personal Computers and IBM-compatible computers. See the reference manuals for your hardware for complete information on BIOS calls.

17.5.2 Defining and Redefining Interrupt Routines

You can write your own interrupt routines, either to replace an existing routine or to use an undefined interrupt number.

■ Syntax

label **PROC FAR**
statements
IRET
label **ENDP**

An interrupt routine can be written like a procedure by using the **PROC** and **ENDP** directives. The only differences are that the routine should always be defined as far and the routine should be terminated by an **IRET** instruction instead of a **RET** instruction.

Your program should replace the address in the interrupt descriptor table with the address of your routine. DOS calls are provided for this task. Another common technique is to jump to the old interrupt routine and let it do the **IRET** instruction. It is usually a good idea to save the old address and restore it before your program ends.

Interrupt routines you may want to replace include the processor's divide-overflow (0h) and overflow (04h) interrupts. You can also replace DOS interrupts such as the critical-error (24h) and CONTROL-C (23h) handlers. Interrupt routines can be part of device drivers. Writing interrupt routines is usually a systems task. The example below illustrates a simple routine. For complete information see the *Microsoft MS-DOS Programmer's Guide* or one of the other reference books on DOS.

80386 Only

The **INT** instruction automatically pushes a 32-bit instruction pointer for 32-bit segments or a 16-bit instruction pointer for 16-bit segments. However, the **IRET** instruction always pops a 16-bit instruction pointer before returning. To pop a 32-bit instruction pointer, you must append the letter **D** (for doubleword) to the instruction to form **IRETD**.

■ Example

```
            .DATA
message     DB      "Overflow - result set to 0",13,10,"$"
vector      DD      ?
            .CODE
start:      mov     ax,@data            ; Load segment location
            mov     ds,ax               ;   into DS register

            mov     ax,3504h            ; Load interrupt 4 and call DOS
            int     21h                 ;   get interrupt vector function
            mov     WORD PTR vector[2],es ; Save segment
            mov     WORD PTR vector[0],bx ;   and offset

            push    ds                  ; Save DS
            mov     ax,cs               ; Load segment of new routine
            mov     ds,ax
            mov     dx,OFFSET overflow  ; Load offset of new routine
            mov     ax,2504h            ; Load interrupt 4 and call DOS
            int     21h                 ;   set interrupt vector function
            pop     ds                  ; Restore
            .
            .
            .
            add     ax,bx               ; Do addition (or multiplication)
            into                        ; Call interrupt 4 if overflow
            .
            .
            .
            lds     dx,vector           ; Load original interrupt address
            mov     ax,2504h            ; Restore interrupt number 4
            int     21h                 ;   with DOS set vector function
            mov     ax,4C00h            ; Terminate function
            int     21h

overflow    PROC    FAR
            sti                         ; Enable interrupts
                                        ;   (turned off by INT)
            mov     ah,09h              ; Display string function
            mov     dx,OFFSET message   ; Load address
            int     21h                 ; Call DOS
            xor     ax,ax               ; Set AX to 0
            xor     dx,dx               ; Set DX to 0
            iret                        ; Return
overflow    ENDP
            .
            END     start
```

In this example, DOS functions are used to save the address of the initial interrupt routine in a variable and to put the address of the new interrupt routine in the interrupt table. Once the new address has been set, the new routine is called any time the interrupt is called. The sample interrupt handler sets the result of a calculation that causes an overflow (either in **AX** or **AX**:**DX**) to 0. It is good practice to restore the original interrupt address before terminating the program.

17.6 Checking Memory Ranges

■ 80186/286/386 Only

Starting with the 80186 processor, the **BOUND** instruction can check to see if a value is within a specified range. This instruction is usually used to check a signed index value to see if it is within the range of an array. **BOUND** is a conditional interrupt instruction like **INTO**. If the condition is not met (the index is out of range), an interrupt 5 is executed.

■ Syntax

BOUND *register16,memory32*
BOUND *register32,memory64* (80386 Only)

To use it for this purpose, the starting and ending values of the array must be stored as 16-bit values in the low and high words of a doubleword memory operand. This operand is given as the source operand. The index value to be checked is given as the destination operand. If the index value is out of range, the instruction issues interrupt 5. This means that the operating system or the program must provide an interrupt routine for interrupt 5. DOS does not provide such a routine, so you must write your own. See Section 17.5, "Using Interrupts," for more information.

■ Example

```
            .DATA
bottom      EQU     0
top         EQU     19
dbounds     LABEL   DWORD               ; Allocate boundaries
wbounds     DW      bottom,top          ;   initialized to bounds
array       DB      top+1 DUP (?)       ; Allocate array
            .CODE
            .
            .
            .                           ; Assume index in DI
            bound   di,dbounds          ; Check to see if it is in range
                                        ;   if out of range, interrupt 5
            mov     dx,array[di]        ; If in range, use it
```

■ 80386 Only

The 80386 can optionally check larger arrays. The destination operand can be a 32-bit register and the source can be a 64-bit memory operand containing 32-bit starting and ending values.

CHAPTER 18

PROCESSING STRINGS

18.1 Setting Up String Operations.....365
18.2 Moving Strings.....368
18.3 Searching Strings.....370
18.4 Comparing Strings.....371
18.5 Filling Strings.....373
18.6 Loading Values from Strings.....374
18.7 Transferring Strings to and from Ports.....375

The 8086-family processors have a full set of instructions for manipulating strings. In the discussion of these instructions, the term "string" refers not only to the common definition of a string—a sequence of bytes containing characters—but to any sequence of bytes or words (or doublewords on the 80386).

The following instructions are provided for 8086-family string functions:

Instruction	Description
MOVS	Moves string from one location to another
SCAS	Scans string for specified values
CMPS	Compares values in one string with values in another
LODS	Loads values from a string to accumulator register
STOS	Stores values from accumulator register to a string
INS	Transfers values from a port to memory
OUTS	Transfers values from memory to a port

All these instructions use registers in the same way and have a similar syntax. Most are used with the repeat instruction prefixes: **REP**, **REPE**, **REPNE**, **REPZ**, and **REPNZ**.

This chapter first explains the general format for string instructions and then tells you how to use each instruction.

18.1 Setting Up String Operations

The string instructions all work in a similar way. Once you understand the general procedure, it is easy to adapt the format for a particular string operation. The five steps are listed below:

1. Make sure the direction flag indicates the direction in which you want the string to be processed. If the direction flag (**DF**) is clear, the string will be processed up (from low addresses to high addresses). If the direction flag is set, the string will be processed down (from high addresses to low addresses). The **CLD** instruction clears the flag, while **STD** sets it. Under DOS, the direction flag will normally be cleared if your program has not changed it.
2. Load the number of iterations for the string instruction into the **CX** register. For instance, if you want to process a 100-byte string, load 100. If a string instruction will be terminated conditionally,

load the maximum number of iterations that can be done without an error.

3. Load the starting offset address of the source string into **DS**:**SI** and the starting address of the destination string into **ES**:**DI**. Some string instructions take only a destination or source (shown in Table 18.1 below). Normally the segment address of the source string should be **DS**, but you can use a segment override with the string instruction to specify a different segment. You cannot override the segment address for the destination string. Therefore you may need to change the value of **ES**.
4. Choose the appropriate repeat-prefix instruction. Table 18.1 shows the repeat prefixes that can be used with each instruction.
5. Put the appropriate string instruction immediately after the repeat prefix (on the same line).

String instructions have two basic forms, as shown below:

■ Syntax 1

[[*repeatprefix*]] *stringinstruction*[[**ES:**[[*destination*,]]]][[[[*segmentregister*:]]*source*]]

The string instruction can be given with the source and/or destination as operands. The size of the operand or operands indicates the size of the objects to be processed by the string. Note that the operands only specify the size. The actual values to be worked on are the ones pointed to by **DS**:**SI** and/or **ES**:**DI**. No error is generated if the operand is not the same as the actual source or destination. One important advantage of this syntax is that the source operand can have a segment override. The destination operand is always relative to **ES** and cannot be overridden.

■ Syntax 2

[[*repeatprefix*]] *stringinstruction***B**
[[*repeatprefix*]] *stringinstruction***W**
[[*repeatprefix*]] *stringinstruction***D** (80386 only)

The letter **B** or **W** appended to the string instruction indicates bytes or words; the letter **D** indicates doublewords on the 80386. With a letter appended to a string instruction, no operand is allowed.

For instance, **MOVS** can be given with byte operands to move bytes or with word operands to move words. As an alternative, **MOVSB** can be given with no operands to move bytes or **MOVSW** can be given with no operands to move words.

Note

Instructions that specify the size in the name never accept operands. Therefore, the following statement is illegal:

```
lodsb   es:0                ; Illegal - no operand allowed
```

Instead, the statement must be coded as shown below:

```
lods    BYTE PTR es:0       ; Legal - use type specifier
```

If a repeat prefix is used, it can be one of the following instructions:

Instruction	Description
REP	Repeats for a specified number of iterations. The number is given in **CX**.
REPE or **REPZ**	Repeats while equal. The maximum number of iterations should be specified in **CX**.
REPNE or **REPNZ**	Repeats while not equal. The maximum number of iterations should be specified in **CX**.

REPE is the same as **REPZ**, and **REPNE** is the same as **REPNZ**. You can use whichever name you find more mnemonic. The prefixes ending with **E** are used in syntax listings and tables in the rest of this chapter.

Table 18.1 lists each string instruction with the type of repeat prefix it uses and whether the instruction works on a source, a destination, or both.

Table 18.1
Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	**DS:SI, ES:DI**
SCAS	**REPE/REPNE**	Destination	**ES:DI**
CMPS	**REPE/REPNE**	Both	**ES:DI, DS:SI**
LODS	None	Source	**DS:SI**
STOS	**REP**	Destination	**ES:DI**
INS	**REP**	Destination	**ES:DI**
OUTS	**REP**	Source	**DS:SI**

At run time, a string instruction preceded by a repeat sequence causes the processor to take the following steps:

1. Checks the **CX** registers and exits from the string instruction if **CX** is 0.
2. Performs the string operation once.
3. Increases **SI** and/or **DI** if the direction flag is cleared. Decreases **SI** and/or **DI** if the direction flag is set. The amount of increase or decrease is one for byte operations, two for word operations, or four for doubleword operations (80386 only).
4. Decrements **CX** (no flags are modified).
5. If the string instruction is **SCAS** or **CMPS**, checks the zero flag and exits if the repeat condition is false—that is, if the flag is set with **REPE** or **REPZ** or if it is clear with **REPNE** or **REPNZ**.
6. Goes to the next iteration (step 1).

Although string instructions (except **LODS**) are most often used with repeat prefixes, they can also be used by themselves. In this case, the **SI** and/or **DI** registers are adjusted as specified by the direction flag and the size of operands. However, you must decrement the **CX** register and set up a loop for the repeated action.

Note

Although you can use a segment override on the source operand, a segment override combined with a repeat prefix can cause problems in certain situations on all processors except the 80386. If an interrupt occurs during the string operation, the segment override is lost and the rest of the string operation processes incorrectly. Segment overrides can be used safely when interrupts are turned off, when a string instruction is used without a segment override, or when a 80386 processor is used.

18.2 Moving Strings

The **MOVS** instruction is used to move data from one area of memory to another.

■ Syntax

[REP] MOVS [ES:]*destination*,[*segmentregister*:]*source*
[REP] MOVSB
[REP] MOVSW
[REP] MOVSD (80386 only)

To move the data, load the count and the source and destination addresses into the appropriate registers, as discussed in Section 18.1, "Setting Up String Operations." Then use the **REP** instruction with the **MOVS** instruction.

■ Example 1

```
            .MODEL  small
            .DATA
source      DB      10 DUP ('0123456789')
destin      DB      100 DUP (?)
            .CODE
            mov     ax,@data            ; Load same segment
            mov     ds,ax               ;   to both DS
            mov     es,ax                   and ES
            .
            .
            .
            cld                         ; Work upward
            mov     cx,100              ; Set iteration count to 100
            mov     si,OFFSET source    ; Load address of source
            mov     di,OFFSET destin    ; Load address of destination
            rep     movsb               ; Move 100 bytes
```

Example 1 shows how to move a string by using string instructions. For comparison, Example 2 shows a much less efficient way of doing the same operation without string instructions.

■ Example 2

```
            .MODEL  small
            .DATA
source      DB      10 DUP ('0123456789')
destin      DB      100 DUP (?)
            .CODE
            .                           ; Assume ES = DS
            .
            .
            mov     cx,100              ; Set iteration count to 100
            mov     si,OFFSET source    ; Load offset of source
            mov     di,OFFSET destin    ; Load offset of destination
repeat:     mov     al,es:[si]          ; Get a byte from source
            mov     [di],al             ; Put it in destination
            inc     si                  ; Increment source pointer
            inc     di                  ; Increment destination pointer
            loop    repeat              ; Do it again
```

Both examples illustrate how to move byte strings in a small-model program in which **DS** already points to the segment containing the variables. In such programs, **ES** can be set to the same value as **DS**.

There are several variations on this. If the source string was not in the current data segment, you could load the starting address of its segment into **ES**. Another option would be to use the **MOVS** instruction with operands and give a segment override on the source operand. For example, you could use the following statement if **ES** pointed to both the source and the destination strings:

```
rep     movs destin,es:source
```

It is sometimes faster to move a string of bytes as words (or as double-words on the 80386). You must adjust for any odd bytes, as shown in Example 3. Assume the source and destination are already loaded.

■ Example 3

```
mov     cx,count            ; Load count
shr     cx,1                ; Divide by 2 (carry will be set
                            ;   if count is odd)
rep     movsw               ; Move words
rcl     cx,1                ; If odd, make CX 1
rep     movsb               ; Move odd byte if there is one
```

18.3 Searching Strings

The **SCAS** instruction is used to scan a string for a specified value.

■ Syntax

[REPE | REPNE] SCAS [ES:] *destination*
[REPE | REPNE] SCASB
[REPE | REPNE] SCASW
[REPE | REPNE] SCASD (80386 only)

SCAS and its variations work only on a destination string, which must be pointed to by **ES:DI**. The value to scan for must be in the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for double-words.

The **SCAS** instruction works by comparing the value pointed to by **DI** with the value in the accumulator. If the values are the same, the zero flag is set. Thus the instruction only makes sense when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first occurrence of a specified value, use the **REPNE** or **REPNZ** instruction. If the value is found, **ES**:**DI** will point to the value immediately after the first occurrence. You can decrement **DI** to make it point to the first matching value.

If you want to search for the first value that does not have a specified value, use **REPE** or **REPZ**. If the value is found, **ES**:**DI** will point to the position after the first nonmatching value. You can decrement **DI** to make it point to the first nonmatching value.

If the value is not found, the **CX** register will contain 0. You can use the **JCXZ** instruction to handle cases where the value is not found.

■ Example

```
            .DATA
string      DB      "The quick brown fox jumps over the lazy dog"
lstring     EQU     $-string            ; Length of string
pstring     DD      string              ; Far pointer to string
            .CODE
            .
            .
            .
            cld                         ; Work upward
            mov     cx,lstring          ; Load length of string
            les     di,pstring          ; Load address of string
            mov     al,'z'              ; Load character to find
            repne   scasb               ; Search
            jcxz    notfound            ; CX is 0 if not found
            .                           ; ES:DI points to character
            .                           ;   after first 'z'
            .
notfound:                               ; Special case for not found
```

This example assumes that **ES** is not the same as **DS**, but that the address of the string is stored in a pointer variable. The **LES** instruction is used to load the far address of the string into **ES**:**DI**.

18.4 Comparing Strings

The **CMPS** instruction is used to compare two strings and point to the address where a match or nonmatch occurs.

■ Syntax

[REPE | REPNE] CMPS [segment register:]source,[ES:],destination
[REPE | REPNE] CMPSB
[REPE | REPNE] CMPSW
[REPE | REPNE] CMPSD (80386 only)

The count and the addresses of the strings are loaded into registers, as described in Section 18.1, "Setting Up String Operations." Either string can be considered the destination or source string unless a segment override is used. Notice that unlike other instructions, **CMPS** requires the source be on the left.

The **CMPS** instruction works by comparing in turn each value pointed to by **DI** with the value pointed to by **SI**. If the values are the same, the zero flag is set. Thus the instruction makes sense only when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first match between the strings, use the **REPNE** or **REPNZ** instruction. If a match is found, **ES**:**DI** and **DS**:**SI** will point to the position after the first match in the respective strings. You can decrement **DI** or **SI** to point to the match.

If you want to search for a nonmatch, use **REPE** or **REPZ**. If a nonmatch is found, **ES**:**DI** and **DS**:**SI** will point to the position after the first nonmatch in the respective strings. You can decrement **DI** or **SI** to point to the nonmatch.

If the specified condition (match or nonmatch) never occurs, the **CX** register will contain zero. You can use the **JCXZ** instruction to handle cases in which the entire string is processed.

■ Example

```
            .MODEL  large
            .DATA
string1     DB      "The quick brown fox jumps over the lazy dog"
            .FARDATA
string2     DB      "The quick brown dog jumps over the lazy fox"
lstring     EQU     $-string2
            .CODE
            mov     ax,@data            ; Load data segment
            mov     ds,ax               ;   into DS
            mov     ax,@fardata         ; Load far data segment
            mov     es,ax               ;   into ES
            .
            .
            .
            cld                         ; Work upward
            mov     cx,lstring          ; Load length of string
            mov     si,OFFSET string1   ; Load offset of string1
            mov     di,OFFSET string2   ; Load offset of string2
            repe    cmpsb               ; Compare
            jcxz    allmatch            ; CX is 0 if no nonmatch
            dec     si                  ; Adjust to point to nonmatch
            dec     di                  ;   in each string
            .
            .
allmatch:   .                           ; Special case for all match
```

This example assumes that the strings are in different segments. Both segments must be initialized to the appropriate segment register.

18.5 Filling Strings

The **STOS** instruction is used to store a specified value in each position of a string.

■ Syntax

[REP] STOS [ES:]*destination*
[REP] STOSB
[REP] STOSW
[REP] STOSD (80386 only)

The string is considered the destination, so it must be pointed to by **ES:DI**. The length and address of the string must be loaded into registers, as described in Section 18.1, "Setting Up String Operations." The value to store must be in the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for doublewords.

For each iteration specified by the **REP** instruction prefix, the value in the accumulator is loaded into the string.

■ Example

```
            .MODEL  small
            .DATA
destin      DB      100 DUP ?
            .CODE
            .                               ; Assume ES = DS
            .
            .
            cld                             ; Work upward
            mov     ax,'aa'                 ; Load character to fill
            mov     cx,50                   ; Load length of string
            mov     di,OFFSET destin        ; Load address of destination
            rep     stosw                   ; Store 'a' into array
```

This example loads 100 bytes containing the character "a." Notice that this is done by storing 50 words rather than 100 bytes. This makes the code faster by reducing the number of iterations. You would have to adjust for the last byte if you wanted to fill an odd number of bytes.

18.6 Loading Values from Strings

The **LODS** instruction is used to load a value from a string into a register.

■ Syntax

LODS [*segmentregister:*]*source*
LODSB
LODSW
LODSD (80386 only)

The string is considered the source, so it must be pointed to by **DS:SI**. The value is always loaded from the string into the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for doublewords.

Unlike other string instructions, **LODS** is not normally used with a repeat prefix since there is no reason to move a value repeatedly to a register. However, **LODS** does adjust the **DI** register as specified by the direction flag and the size of operands. The programmer must code the instructions to use the value after it is loaded.

■ Example 1

```
            .DATA
stuff       DB      0,1,2,3,4,5,6,7,8,9
            .CODE
            .
            .
            .
            cld                         ; Work upward
            mov     cx,10               ; Load length
            mov     si,OFFSET stuff     ; Load offset of source
            mov     ah,2                ; Display character function
get:        lodsb                       ; Get a character
            add     al,48               ; Convert to ASCII
            mov     dl,al               ; Move to DL
            int     21h                 ; Call DOS to display character
            loop    get                 ; Repeat
```

Example 1 loads, processes, and displays each byte in a string of bytes.

■ Example 2

```
            .DATA
buffer      DB      80 DUP(?)           ; Create buffer for argument string
            .CODE
start:      mov     ax,@data            ; Initialize DS
            mov     ds,ax
                                        ; On start-up ES points to PSP
            cld                         ; Work upward
            mov     cl,BYTE PTR es:[80h];Load length of arguments
            xor     ch,ch
            mov     di,OFFSET buffer    ; Load offset of buffer
            mov     si,82h              ; Load position of argument string
            mov     dx,es               ; Exchange ES and DS
            mov     ax,ds
            mov     es,ax
            mov     ds,dx
another:    lodsb                       ; Get a character
            cmp     al,'a'              ; Is it high enough to be upper?
            jb      noway               ; No? Check
            cmp     al,'z'              ; Is it low enough to be letter?
            ja      noway
            sub     al,32               ; Yes? Convert to uppercase
noway:      stosb
            loop    another             ; Repeat
            mov     dx,es               ; Restore ES and DS
            mov     ax,ds
            mov     es,ax
            mov     ds,dx
```

Example 2 copies the command arguments from position 82h in the DOS Program Segment Prefix (PSP) while converting them to uppercase. See the *Microsoft MS-DOS Programmer's Reference* or one of the many other books on DOS for information about the PSP. Notice that both **LODSB** and **STOSB** are used without repeat prefixes.

18.7 Transferring Strings to and from Ports

■ 80186/286/386 Only

The **INS** instruction reads a string from a port to memory, and the **OUTS** instruction writes a string from memory to a port.

■ Syntax

OUTS DX,[*segmentregister:*]*source*
OUTSB
OUTSW
OUTSD (80386 only)

INS [**ES:**] *destination*,**DX**
INSB
INSW
INSD (80386 only)

The **INS** and **OUTS** instructions require that the number of the port be in **DX**. The port cannot be specified as an immediate value, as it can be with **IN** and **OUT**.

To move the data, load the count into **CX**. The string to be transferred by **INS** is considered the destination string, so it must be pointed to by **ES:DI**. The string to be transferred by **OUTS** is considered the source string, so it must be pointed to by **DS:SI**.

If you specify the source or destination as an operand, **DX** must be specified. Otherwise **DX** is assumed and should be omitted.

If you need to process the string as it is transferred (for instance, to check for the end of a null-terminated string), you must set up the loop yourself instead of using the **REP** instruction prefix.

■ Example

```
            .DATA
count       EQU     100
buffer      DB      count DUP (?)
inport      DW      ?
            .CODE
            .                               ; Assume ES = DS
            .
            .
            cld                             ; Work upward
            mov     cx,count                ; Load length to transfer
            mov     di,OFFSET buffer        ; Load address of destination
            mov     dx,inport               ; Load port number
            rep     insb                    ; Transfer the string
                                            ;    from port to buffer
```

CHAPTER 19

CALCULATING WITH A MATH COPROCESSOR

19.1 Coprocessor Architecture 379
19.1.1 Coprocessor Data Registers 380
19.1.2 Coprocessor Control Registers 381
19.2 Emulation 382
19.3 Using Coprocessor Instructions 382
19.3.1 Using Implied Operands in the Classical-Stack Form 383
19.3.2 Using Memory Operands 384
19.3.3 Specifying Operands in the Register Form 385
19.3.4 Specifying Operands in the Register-Pop Form 386
19.4 Coordinating Memory Access 387
19.5 Transferring Data 388
19.5.1 Transferring Data to and from Registers 388
19.5.2 Loading Constants 391
19.5.3 Transferring Control Data 392
19.6 Doing Arithmetic Calculations 393
19.7 Controlling Program Flow 399
19.7.1 Comparing Operands to Control Program Flow 401
19.7.2 Testing Control Flags after Other Instructions 404
19.8 Using Transcendental Instructions 404
19.9 Controlling the Coprocessor 406

The 8087-family coprocessors are used to do fast mathematical calculations. When used with real numbers, packed BCD numbers, or long integers, they do calculations many times faster than the same operations done with 8086-family processors.

This chapter explains how to use the 8087-family processors to transfer and process data. The approach taken is from an applications standpoint. Features that would be used by systems programmers (such the flags used when writing exception handlers) are not explained. This chapter is intended as a reference, not a tutorial.

Note

This manual does not attempt to explain the mathematical concepts involved in using certain coprocessor features. It assumes that you will not need to use a feature unless you understand the mathematics involved. For example, you need to understand logarithms to use the **FYL2X** and **FYL2XP1** instructions.

19.1 Coprocessor Architecture

The math coprocessor works simultaneously with the main processor. However, since the coprocessor cannot handle device input or output, most data originates in the main processor.

The main processor and the coprocessor each have their own registers, which are completely separate and inaccessible to the other. They exchange data through memory, since memory is available to both.

Ordinarily you follow these three steps when using the coprocessor:

1. Load data from memory to coprocessor registers
2. Process the data
3. Store the data from coprocessor registers back to memory

Step 2, processing the data, can occur while the main processor is handling other tasks. Steps 1 and 3 must be coordinated with the main processor so that the processor and coprocessor do not try to access the same memory at the same time, as is explained in Section 19.4, "Transferring Data."

19.1.1 Coprocessor Data Registers

The 8087-family coprocessors have eight 80-bit data registers. Unlike 8086-family registers, the coprocessor data registers are organized as a stack. As data is pushed into the top register, previous data items move into higher-numbered registers. Register 0 is the top of the stack; register 7 is the bottom. The syntax for specifying registers is shown below:

ST[[(*number*)]]

The *number* must be a digit between 0 and 7. If *number* is omitted, register 0 (top of stack) is assumed.

All coprocessor data are stored in registers in the temporary-real format. This is the 10-byte IEEE format described in Section 6.3.1.5, "Real-Number Variables." The registers and the register format are shown in Figure 19.1.

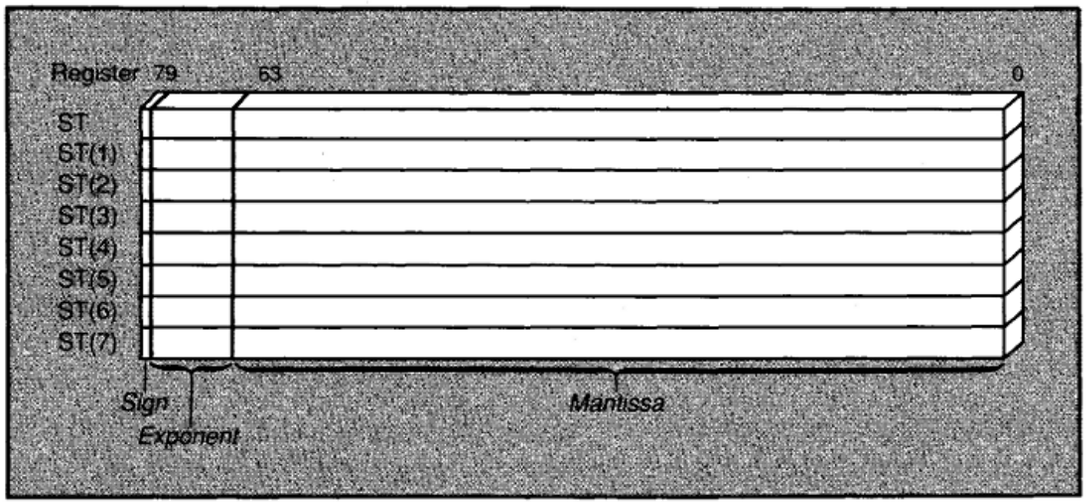


Figure 19.1 Coprocessor Data Registers

Internally, all calculations are done on numbers of the same type. Since temporary-real numbers have the greatest precision, lower-precision numbers are guaranteed not to lose precision as a result of calculations. The instructions that transfer values between the main processor and the coprocessor automatically convert numbers to and from the temporary-real format.

19.1.2 Coprocessor Control Registers

The 8087-family coprocessors have seven 16-bit control registers. The most useful control registers are made up of bit fields or flags. Some flags control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086-family flags registers.

You do not need to understand these registers to do most coprocessor operations. Control flags are set by default to the values appropriate for most programs. Errors and exceptions are reported in the status-word register. However, the coprocessor already has a default system for handling exceptions. Applications programmers can usually accept the defaults. Systems programmers may want to use the status-word and control-word registers when writing exception handlers, but such problems are beyond the scope of this manual.

Figure 19.2 shows the overall layout of the control registers including the control word, status word, tag word, instruction pointer, and operand pointer. The format of each of the registers is not shown, since these registers are generally of use only to systems programmers. The exception is the condition-code bits of the status-word register. These bits are explained in Section 19.7, "Controlling Program Flow."

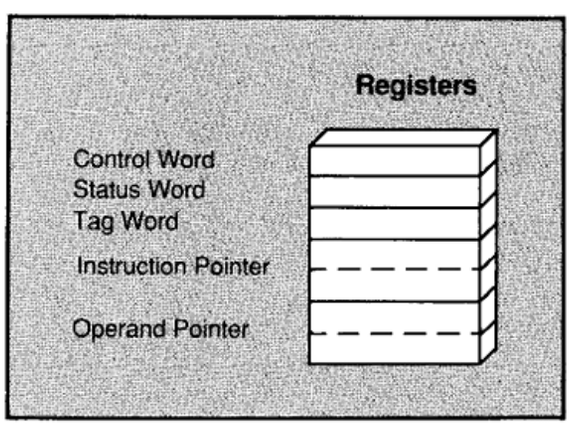


Figure 19.2 Coprocessor Control Registers

The control registers are explained in more detail in the *Microsoft Macro Assembler Reference.*

19.2 Emulation

If you have a Microsoft high-level language that supports floating-point emulation, you can write assembly-language procedures that use the emulator library when called from the high-level language. First write the procedure by using coprocessor instructions, then assemble it using the **/E** option, and finally link it with your high-level language modules. When compiling modules, use the compiler options that specify emulation.

Some coprocessor instructions are not emulated by Microsoft emulation libraries. How unemulated instructions vary depends on the language and version. If you use a coprocessor instruction that is not emulated, the program will generate a run-time error when it tries to execute the unemulated instruction. You cannot use a Microsoft emulation library with stand-alone assembler programs, since the library depends on the compiler start-up code.

See Section 2.4.5, "Creating Code for a Floating-Point Emulator," for information on the **/E** option. See the *Microsoft Mixed-Language Programming Guide* for information on writing assembly-language procedures for high-level languages.

19.3 Using Coprocessor Instructions

Coprocessor instructions are readily recognizable because, unlike all 8086-family instruction mnemonics, they start with the letter **F.**

Most coprocessor instructions have two operands, but in many cases one or both operands are implied. Often, one operand can be a memory operand; in this case, the other operand is always implied as the stack-top register. Coprocessor instructions can never have immediate operands, and with the exception of the **FSTSW** instruction (see Section 19.5.2, "Loading Constants"), they cannot have processor registers as operands. As with 8086-family instructions, memory-to-memory operations are never allowed. One operand must be a coprocessor register.

Instructions usually have a source and a destination operand. The source specifies one of the values to be processed. It is never changed by the operation. The destination specifies the value to be operated on and replaced with the result of the operation. If operands are specified, the first is the destination and the second is the source.

The stack organization of registers gives the programmer flexibility to think of registers either as elements on a stack or as registers much like 8086-family registers. Table 19.1 lists the variations of coprocessor instructions along with the syntax for each.

Table 19.1

Coprocessor Operand Forms

Instruction Form	Syntax	Implied Operands	Example
Classical-stack	**F***action*	**ST(1),ST**	`fadd`
Memory	**F***action memory*	**ST**	`fadd memloc`
Register	**F***action* **ST(***num***),ST** **F***action* **ST,ST(***num***)**		`fadd st(5),st` `fadd st,st(3)`
Register pop	**F***action***P ST(***num***),ST**		`faddp st(4),st`

Not all instructions accept all operand variations. For example, load and store instructions always require the memory form. Load-constant instructions always take the classical-stack form. Arithmetic instructions can usually take any form.

Some instructions that accept the memory form can have the letter **I** (integer) or **B** (BCD) following the initial **F** to specify how a memory operand is to be interpreted. For example, **FILD** interprets its operand as an integer and **FBLD** interprets its operand as a BCD number. If no type letter is included in the instruction name, the instruction works on real numbers.

19.3.1 Using Implied Operands in the Classical-Stack Form

The classical-stack form treats coprocessor registers like items on a stack. Items are pushed onto or popped off the top elements of the stack. Since only the top item can be accessed on a traditional stack, there is no need to specify operands. The first register (and the second if there are two operands) is always assumed.

In arithmetic operations (see Section 19.6), the top of the stack (**ST**) is the source operand, and the second register (**ST(1)**) is the destination. The result of the operation goes into the destination operand, and the source is popped off the stack. The effect is that both of the values used in the operation are destroyed and the result is left at the top of the stack.

Instructions that load constants always use the stack form (see Section 19.5.1, "Transferring Data to and from Registers"). In this case the constant created by the instruction is the implied source, and the top of the stack (**ST**) is the destination. The source is pushed into the destination.

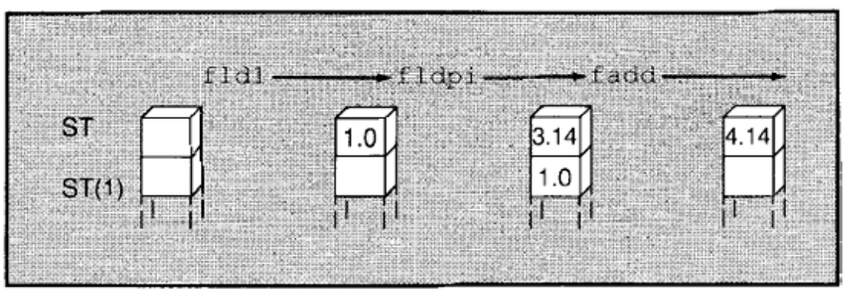
Note

The classical-stack form with its implied operands is similar to the register-pop form, not to the register form. For example, `fadd`, with the implied operands **ST(1),ST**, is equivalent to `faddp st(1),st`, rather than to `fadd st(1),st`.

■ **Example**

```
fld1                    ; Push 1 into first position
fldpi                   ; Push pi into first position
fadd                    ; Add pi and 1 and pop
```

The status of the register stack after each instruction is shown below:



19.3.2 Using Memory Operands

The memory form treats coprocessor registers like items on a stack. Items are pushed from memory onto the top element of the stack, or popped from the top element to memory. Since only the top item can be accessed on a traditional stack, there is no need to specify the stack operand. The top register (**ST**) is always assumed. However, the memory operand must be specified.

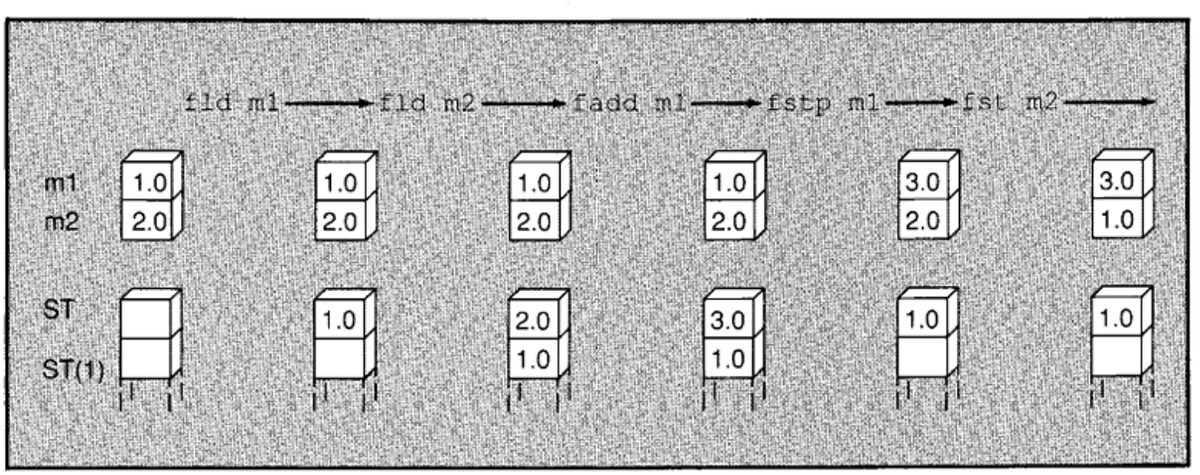
Memory operands can be used in load and store instructions (see Section 19.5.1, "Transferring Data to and from Registers"). Load instructions push source values from memory to an implied destination register (**ST**). Store instructions pop source values from an implied source register (**ST**) to the destination in memory. Some versions of store instructions pop the register stack so that the source is destroyed. Others simply copy the source without changing the stack.

Memory operands can also be used in calculation instructions that operate on two values (see Section 19.6, "Doing Arithmetic Calculations"). The memory operand is always the source. The stack top (**ST**) is always the implied destination. The result of the operation replaces the destination without changing its stack position.

■ Example

```
            .DATA
m1          DD      1.0
m2          DD      2.0
            .CODE
            .
            .
            .
            fld     m1          ; Push m1 into first position
            fld     m2          ; Push m2 into first position
            fadd    m1          ; Add m2 to first position
            fstp    m1          ; Pop first position into m1
            fst     m2          ; Copy first position to m2
```

The status of the register stack and the memory locations used in the instructions is shown below:



19.3.3 Specifying Operands in the Register Form

The register form treats coprocessor registers as traditional registers. Registers are specified the same as 8086-family instructions with two register operands. The only limitation is that one of the two registers must be the stack top (**ST**).

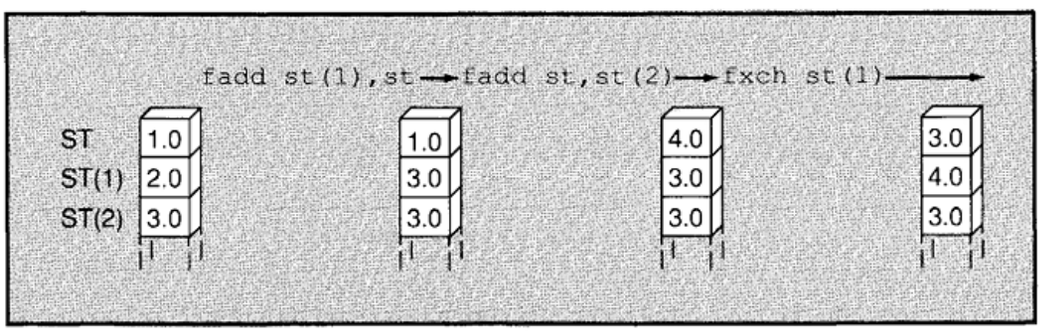
In the register form, operands are specified by name. The second operand is the source; it is not affected by the operation. The first operand is the destination; its value is replaced with the result of the operation. The stack position of the operands does not change.

The register form can only be used with the **FXCH** instruction and with arithmetic instructions that do calculations on two values. With the **FXCH** instruction, the stack top is implied and need not be specified.

■ **Example**

```
fadd    st(1),st    ; Add second position to first -
                    ;   result goes in second position
fadd    st,st(2)    ; Add first position to second -
                    ;   result goes in first position
fxch    st(1)       ; Exchange first and second positions
```

The status of the register stack if the registers were previously initialized to 1.0, 2.0, and 3.0 is shown below:



19.3.4 Specifying Operands in the Register-Pop Form

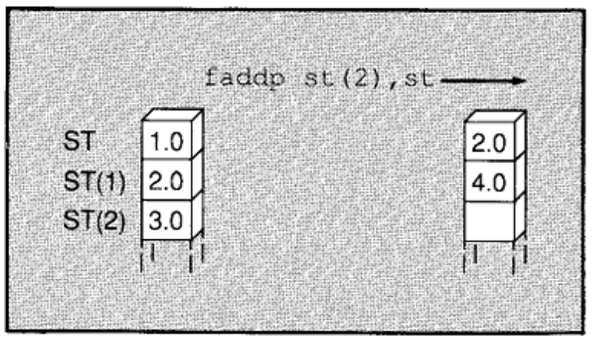
The register-pop form treats coprocessor registers as a modified stack. This form has some of the aspects of both a stack and registers. The destination register can be specified by name, but the source register must always be the stack top.

The result of the operation will be placed in the destination operand, and the stack top will be popped off the stack. The effect is that both values being operated on will be destroyed and the result of the operation will be saved in the specified destination register. The register-pop form is only used for instructions that do calculations on two values.

■ **Example**

```
faddp   st(2),st    ; Add first and third positions and pop -
                    ;   first position destroyed
                    ;   third moves to second and holds result
```

The status of the register stack if the registers were already initialized to 1.0, 2.0, and 3.0 is shown below:



19.4 Coordinating Memory Access

Problems of coordinating memory access can occur when the coprocessor and the main processor both try to access a memory location at the same time. Since the processor and coprocessor work independently, they may not finish working on memory in the order in which you give instructions. There are two separate cases, and they are handled in different ways.

In the first case, if a processor instruction is given and then followed by a coprocessor instruction, the coprocessor must wait until the processor is finished before it can start the next instruction. This is handled automatically by **MASM** for the 8088 and 8086 or by the processor for the 80186, 80286, and 80386.

Coprocessor Differences

To synchronize operations between the 8088 or 8086 processor and the 8087 coprocessor, each 8087 instruction must be preceded by a **WAIT** instruction. This is not necessary for the 80287 or 80387. If you use the **.8087** directive, **MASM** inserts **WAIT** instructions automatically. However, if you use the **.286** or **.386** directive, **MASM** assumes the instructions are for the 80287 or 80387 and does not insert the **WAIT** instructions. If your code will never need to run on an 8086 or 8088 processor, you can make your programs shorter and more efficient by using the **.286** or **.386** directive.

In the second case, if a coprocessor instruction that accesses memory is followed by a processor instruction attempting to access the same memory location, memory access is not automatically synchronized. For instance, if you store a coprocessor register to a variable and then try to load that variable into a processor register, the coprocessor may not be finished. Thus the processor gets the value that was in memory before the coprocessor finished rather than the value stored by the coprocessor. Use the

WAIT or **FWAIT** instruction (they are mnemonics for the same instruction) to ensure that the coprocessor finishes before the processor begins.

■ Example

```
; Coprocessor instruction first - Wait needed

            fist    mem32               ; Store to memory
            fwait                       ; Wait until coprocessor is done
            mov     ax,WORD PTR mem32   ; Move to register
            mov     dx,WORD PTR mem32[2]

; Processor instruction first - No wait needed
            mov     WORD PTR mem32,ax   ; Load memory
            mov     WORD PTR mem32[2],dx
            fild    mem32               ; Load to register
```

19.5 Transferring Data

The 8087-family coprocessors have separate instructions for each of the following types of transfers:

- Transferring data between memory and registers, or between different registers
- Loading certain common constants into registers
- Transferring control data to and from memory

19.5.1 Transferring Data to and from Registers

Data-transfer instructions transfer data between main memory and the coprocessor registers, or between different coprocessor registers. Two basic principles govern data transfers:

- The instruction determines whether a value in memory will be considered an integer, a BCD number, or a real number. The value is always considered a temporary-real number once it is transferred to the coprocessor.
- The size of the operand determines the size of a value in memory. Values in the coprocessor always take up 10 bytes.

The adjustments between formats are made automatically. Notice that floating-point numbers must be stored in the IEEE format, not in the Microsoft Binary format. Data is automatically stored correctly by default. It is stored incorrectly and the coprocessor instructions disabled if

you use the **.MSFLOAT** directive. Data formats for real numbers are explained in Section 6.3.1.5, "Real-Number Variables."

Data are transferred to stack registers by using load commands. These push data onto the stack from memory or coprocessor registers. Data are removed by using store commands. Some store commands pop data off the register stack into memory or coprocessor registers, whereas others simply copy the data without changing it on the stack.

Real Transfers

The following instructions are available for transferring real numbers.

Syntax	Description
FLD *mem*	Pushes a copy of *mem* into **ST**. The source must a 4-, 8-, or 10-byte memory operand. It is automatically converted to the temporary-real format.
FLD ST(*num***)**	Pushes a copy of the specified register into **ST**.
FST *mem*	Copies **ST** to *mem* without affecting the register stack. The destination can be a 4- or 8-byte memory operand. It is automatically converted from temporary-real format to short real or long real format, depending on the size of the operand. It cannot be converted to the 10-byte-real format.
FST ST(*num***)**	Copies **ST** to the specified register. The current value of the specified register is replaced.
FSTP *mem*	Pops a copy of **ST** into *mem*. The destination can be a 4-, 8-, or 10-byte memory operand. It is automatically converted from temporary-real format to the appropriate real-number format, depending on the size of the operand.
FSTP ST(*num***)**	Pops **ST** into the specified register. The current value of the specified register is replaced.
FXCH [[ST(*num***)]]**	Exchanges the value in **ST** with the value in **ST(***num***)**. If no operand is specified, **ST(0)** and **ST(1)** are exchanged.

Integer Transfers

The following instructions are available for transferring binary integers.

Syntax	Description
FILD *mem*	Pushes a copy of *mem* into **ST**. The source must be a 2-, 4-, or 8-byte integer memory operand. It is interpreted as an integer and converted to temporary-real format.
FIST *mem*	Copies **ST** to *mem*. The destination must be a 2- or 4-byte memory operand. It is automatically converted from temporary-real format to a word or a doubleword, depending on the size of the operand. It cannot be converted to a quadword integer.
FISTP *mem*	Pops **ST** into *mem*. The destination must be a 2-, 4-, or 8-byte memory operand. It is automatically converted from temporary-real format to a word, doubleword, or quadword integer, depending on the size of the operand.

Packed BCD Transfers

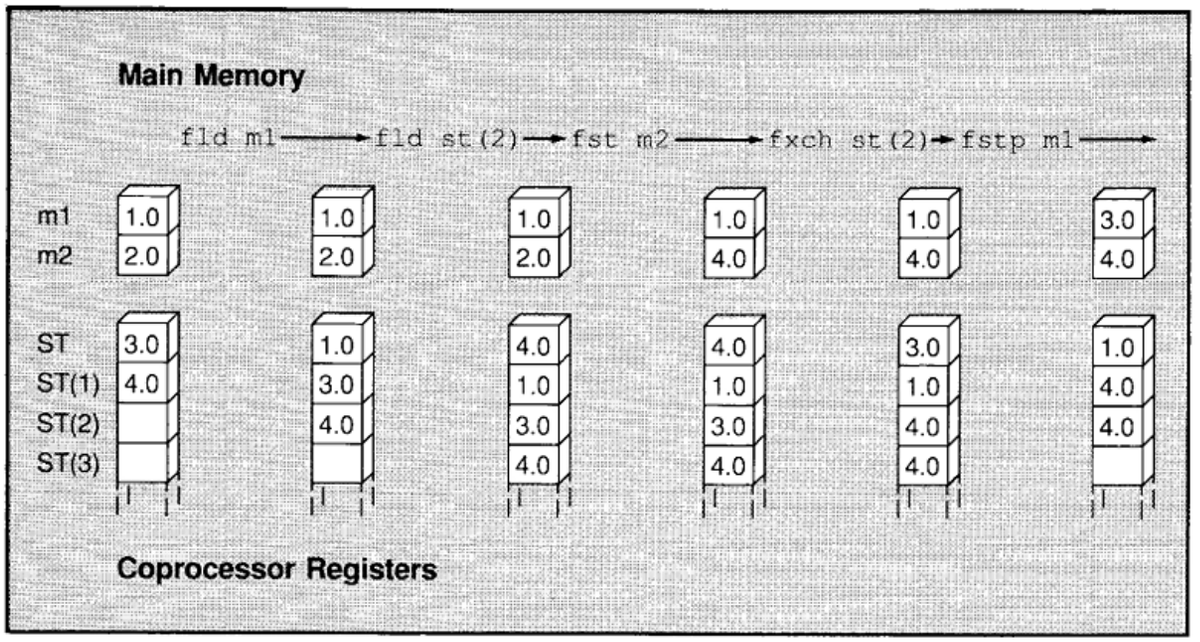
The following instructions are available for transferring BCD integers.

Syntax	Description
FBLD *mem*	Pushes a copy of *mem* into **ST**. The source must be a 10-byte memory operand. It should contain a packed BCD value, although no check is made to see that the data is valid.
FBSTP *mem*	Pops **ST** into *mem*. The destination must be a 10-byte memory operand. The value is rounded to an integer if necessary, and converted to a packed BCD value.

■ Example 1

```
fld     m1              ; Push m1 into first item
fld     st(2)           ; Push third item into first
fst     m2              ; Copy first item to m2
fxch    st(2)           ; Exchange first and third items
fstp    m1              ; Pop first item into m1
```

With the assumption that registers **ST** and **ST(1)** were previously initialized to 3.0 and 4.0, the status of the register stack is shown below:



■ Example 2

```
                .DATA
shortreal       DD      100 DUP (?)
longreal        DQ      100 DUP (?)
                .CODE
                .                               ; Assume array shortreal has been
                .                               ;   filled by previous code
                .
                mov     cx,100                  ; Initialize loop
                xor     si,si                   ; Clear pointer into shortreal
                xor     di,di                   ; Clear pointer into longreal
again:          fld     shortreal[si]           ; Push shortreal
                fstp    longreal[di]            ; Pop longreal
                add     si,4                    ; Increment source pointer
                add     di,8                    ; Increment destination pointer
                loop    again                   ; Do it again
```

Example 2 illustrates one way of doing run-time type conversions.

19.5.2 Loading Constants

Constants cannot be given as operands and loaded directly into coprocessor registers. You must allocate memory and initialize the variable to a constant value. The variable can then be loaded by using one of the load instructions described in Section 19.5.1, "Transferring Data to and from Registers."

However, special instructions are provided for loading certain constants. You can load 0, 1, pi, and several common logarithmic values directly. Using these instructions is faster and often more precise than loading the values from initialized variables.

The instructions that load constants all have the stack top as the implied destination operand. The constant to be loaded is the implied source operand. The instructions are listed below.

Syntax	Description
FLDZ	Pushes 0 into **ST**
FLD1	Pushes 1 into **ST**
FLDPI	Pushes the value of pi into **ST**
FLDL2E	Pushes the value of $\log_2 e$ into **ST**
FLDL2T	Pushes $\log_2 10$ into **ST**
FLDLG2	Pushes $\log_{10} 2$ into **ST**
FLDLN2	Pushes $\log_e 2$ **ST**

19.5.3 Transferring Control Data

The coprocessor data area, or parts of it, can be stored to memory and later loaded back. One reason for doing this is to save a snapshot of the coprocessor state before going into a procedure, and restore the same status after the procedure. Another reason is to modify coprocessor behavior by storing certain data to main memory, operating on the data with 8086-family instructions, and then loading it back to the coprocessor data area.

You can choose to transfer the entire coprocessor data area, the control registers, or just the status or control word. Applications programmers seldom need to load anything other than the status word.

All the control-transfer instructions take a single memory operand. Load instructions use the memory operand as the destination; store instructions use it as the source. The coprocessor data area is the implied source for load instructions and the implied destination for store instructions.

Each store instruction has two forms. The "wait form" checks for unmasked numeric-error exceptions and waits until they have been handled. The "no-wait" form (which always begins with **FN**) ignores unmasked exceptions. The instructions are listed below.

Syntax	Description
FLDCW *mem2byte*	Loads control word
F[N]STCW *mem2byte*	Stores control word
F[N]STSW *mem2byte*	Stores status word
FLENV *mem14byte*	Loads environment
F[N]STENV *mem14byte*	Stores environment
FRSTOR *mem94byte*	Restores state
F[N]SAVE *mem94byte*	Saves state

■ 80287/387 Only

Starting with the 80287, the **FSTSW** and **FNSTSW** instructions can store data directly to the **AX** register. This is the only case in which data can be transferred directly between processor and coprocessor registers, as shown below:

```
fstsw   ax
```

■ 80387 Only

In 32-bit mode, the 80387 stores 32-bit addresses in the instruction and operand pointers. Therefore, the **FSAVE** instruction stores 98 bytes instead of 94, and the **FSTENV** instruction stores 18 bytes instead of 14.

19.6 Doing Arithmetic Calculations

The math coprocessors offer a rich set of instructions for doing arithmetic. Most arithmetic instructions accept operands in any of the formats discussed in Section 19.3, "Using Coprocessor Instructions."

When using memory operands with an arithmetic instruction, make sure you indicate in the name whether you want the memory operand to be treated as a real number or an integer. For example, use **FADD** to add a real number to the stack top or **FIADD** to add an integer to the stack top. You do not need to specify the operand type in the instruction if both operands are stack registers, since register values are always real numbers.

You cannot do arithmetic on BCD numbers in memory. You must use **FBLD** to load the numbers into stack registers.

The arithmetic instructions are listed below.

Addition

The following instructions add the source and destination and put the result in the destination.

Syntax	Description
FADD	Classical-stack form. Adds **ST** and **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FADD ST(*num*),ST	Register form with stack top as source. Adds the two register values and replaces **ST(*num*)** with the result.
FADD ST,ST(*num*)	Register form with stack top as destination. Adds the two register values and replaces **ST** with the result.
FADD *mem*	Real-memory form. Adds a real number in *mem* to **ST**. The result replaces **ST**.
FIADD *mem*	Integer-memory form. Adds an integer in *mem* to **ST**. The result replaces **ST**.
FADDP ST(*num*),ST	Register-pop form. Adds the two register values and pops the result into **ST(*num*)**. Both operands are destroyed.

Normal Subtraction

The following instructions subtract the source from the destination and put the difference in the destination. Thus the number being subtracted from is replaced by the result.

Syntax	Description
FSUB	Classical-stack form. Subtracts **ST** from **ST(1)** and pops the result into **ST**. Both operands are destroyed.

FSUB ST(*num*),ST	Register form with stack top as source. Subtracts **ST** from **ST(*num*)** and replaces **ST(*num*)** with the result.
FSUB ST,ST(*num*)	Register form with stack top as destination. Subtracts **ST(*num*)** from **ST** and replaces **ST** with the result.
FSUB *mem*	Real-memory form. Subtracts the real number in *mem* from **ST**. The result replaces **ST**.
FISUB *mem*	Integer-memory form. Subtracts the integer in *mem* from **ST**. The result replaces **ST**.
FSUBP ST(*num*),ST	Register-pop form. Subtracts **ST** from **ST(*num*)** and pops the result into **ST(*num*)**. Both operands are destroyed.

Reversed Subtraction

The following instructions subtract the destination from the source and put the difference in the destination. Thus the number subtracted is replaced by the result.

Syntax	Description
FSUBR	Classical-stack form. Subtracts **ST(1)** from **ST** and pops the result into **ST**. Both operands are destroyed.
FSUBR ST(*num*),ST	Register form with stack top as source. Subtracts **ST(*num*)** from **ST** and replaces **ST(*num*)** with the result.
FSUBR ST,ST(*num*)	Register form with stack top as destination. Subtracts **ST** from **ST(*num*)** and replaces **ST** with the result.
FSUBR *mem*	Real-memory form. Subtracts **ST** from the real number in *mem*. The result replaces **ST**.
FISUBR *mem*	Integer-memory form. Subtracts **ST** from the integer in *mem*. The result replaces **ST**.
FSUBRP ST(*num*),ST	Register-pop form. Subtracts **ST(*num*)** from **ST** and pops the result into **ST(*num*)**. Both operands are destroyed.

Multiplication

The following instructions multiply the source and destination and put the product in the destination.

Syntax	Description
FMUL	Classical-stack form. Multiplies **ST** by **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FMUL ST(*num***),ST**	Register form with stack top as source. Multiplies the two register values and replaces **ST(***num***)** with the result.
FMUL ST,ST(*num***)**	Register form with stack top as destination. Multiplies the two register values and replaces **ST** with the result.
FMUL *mem*	Real-memory form. Multiplies a real number in *mem* by **ST**. The result replaces **ST**.
FIMUL *mem*	Integer-memory form. Multiplies an integer in *mem* by **ST**. The result replaces **ST**.
FMULP ST(*num***),ST**	Register-pop form. Multiplies the two register values and pops the result into **ST(***num***)**. Both operands are destroyed.

Normal Division

The following instructions divide the destination by the source and put the quotient in the destination. Thus the dividend is replaced by the quotient.

Syntax	Description
FDIV	Classical-stack form. Divides **ST(1)** by **ST** and pops the result into **ST**. Both operands are destroyed.
FDIV ST(*num***),ST**	Register form with stack top as source. Divides **ST(***num***)** by **ST** and replaces **ST(***num***)** with the result.
FDIV ST,ST(*num***)**	Register form with stack top as destination. Divides **ST** by **ST(***num***)** and replaces **ST** with the result.

FDIV *mem*	Real-memory form. Divides **ST** by the real number in *mem*. The result replaces **ST**.
FIDIV *mem*	Integer-memory form. Divides **ST** by the integer in *mem*. The result replaces **ST**.
FDIVP ST(*num***),ST**	Register-pop form. Divides **ST(***num***)** by **ST** and pops the result into **ST(***num***)**. Both operands are destroyed.

Reversed Division

The following instructions divide the source by the destination and put the quotient in the destination. Thus the divisor is replaced by the quotient.

Syntax	Description
FDIVR	Classical-stack form. Divides **ST** by **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FDIVR ST(*num***),ST**	Register form with stack top as source. Divides **ST** by **ST(***num***)** and replaces **ST(***num***)** with the result.
FDIVR ST,ST(*num***)**	Register form with stack top as destination. Divides **ST(***num***)** by **ST** and replaces **ST** with the result.
FDIVR *mem*	Real-memory form. Divides the real number in *mem* by **ST**. The result replaces **ST**.
FIDIVR *mem*	Integer-memory form. Divides the integer in *mem* by **ST**. The result replaces **ST**.
FDIVRP ST(*num***),ST**	Register-pop form. Divides **ST** by **ST(***num***)** and pops the result into **ST(***num***)**. Both operands are destroyed.

Other Operations

The following instructions all use the stack top (**ST**) as an implied destination operand. The result of the operation replaces the value in the stack top. No operand should be given.

Syntax	Description
FABS	Sets the sign of **ST** to positive.
FCHS	Reverses the sign of **ST**.
FRNDINT	Rounds the **ST** to an integer.
FSQRT	Replaces the contents of **ST** with its square root.
FSCALE	Scales by powers of two by adding the value of **ST(1)** to the exponent of the value in **ST**. This effectively multiplies the stack-top value by two to the power contained in **ST(1)**. Since the exponent field is an integer, the value in **ST(1)** should normally be an integer.
FPREM	Calculates the partial remainder by performing modulo division on the top two stack registers. The value in **ST** is divided by the value in **ST(1)**. The remainder replaces the value in **ST**. The value in **ST(1)** is unchanged. Since this instruction works by repeated subtractions, it can take a lot of execution time if the operands are greatly different in magnitude. **FPREM** is sometimes used with trigonometric functions.
FXTRACT	Breaks a number down into its exponent and mantissa and pushes the mantissa onto the register stack. Following the operation, **ST** contains the value of the original mantissa and **ST(1)** contains the value of the unbiased exponent.

80387 Only

The 80387 has a new instruction called **FPREM1**. Its effect is similar to that of **FPREM**, but it conforms to the IEEE standard. The difference between the two instructions is explained in the *Microsoft Macro Assembler Reference*.

■ Example

```
                .DATA
a               DD      3.0
b               DD      7.0
c               DD      2.0
posx            DD      0.0
negx            DD      0.0

                .CODE
                .
                .
                .
; Solve quadratic equation - no error checking

                fld1                    ; Get constants 2 and 4
                fadd    st,st           ; 2 at bottom
                fld     st              ; Copy it
                fmul    a               ; = 2a

                fmul    st(1),st        ; = 4a
                fxch                    ; Exchange
                fmul    c               ; = 4ac

                fld     b               ; Load b
                fmul    st,st           ; = b^2
                fsubr                   ; = b^2 - 4ac
                                        ; Negative value here produces error
                fsqrt                   ; = square root(b^2 - 4ac)
                fld     b               ; Load b
                fchs                    ; Make it negative
                fxch                    ; Exchange
                fld     st              ; Copy square root
                fadd    st,st(2)        ; Plus version = -b + root((b^2 - 4ac)
                fxch                    ; Exchange
                fsubp   st(2),st        ; Minus version = -b - root((b^2 - 4ac)

                fdiv    st,st(2)        ; Divide plus version
                fstp    posx            ; Store it
                fdivr                   ; Divide minus version
                fstp    negx            ; Store it
```

This example solves quadratic equations. It does no error checking and fails for some values because it attempts to find the square root of a negative number. You could enhance the code by using the **FTST** instruction (see Section 19.7.1, "Comparing Operands to Control Program") to check for a negative number or 0 just before the square root is calculated. If `b` squared minus `4ac` is negative or 0, the code can jump to routines that handle special cases for no solution or one solution, respectively.

19.7 Controlling Program Flow

The math coprocessors have several instructions that set control flags in the status word. The 8087-family control flags can be used with conditional jumps to direct program flow in the same way that 8086-family flags are used.

Since the coprocessor does not have jump instructions, you must transfer the status word to memory so that the flags can be used by 8086-family instructions.

An easy way to use the status word with conditional jumps is to move its upper byte into the lower byte of the processor flags. For example, use the following statements:

```
fstsw   mem16       ; Store status word in memory
fwait               ; Make sure coprocessor is done
mov     ax,mem16    ; Move to AX
sahf                ; Store upper word in flags
```

As noted in Section 19.5.3, "Transferring Control Data," you can save several steps by loading the status word directly to **AX** on the 80287 and 80387.

Figure 19.3 shows how the coprocessor control flags line up with the processor flags. **C3** overwrites the zero flag, **C2** overwrites the parity flag, and **C0** overwrites the carry flag. **C1** overwrites an undefined bit, so it cannot be used directly with conditional jumps, although you can use the **TEST** instruction to check **C1** in memory or in a register. The sign and auxiliary-carry flags are also overwritten, so you cannot count on them being unchanged after the operation.

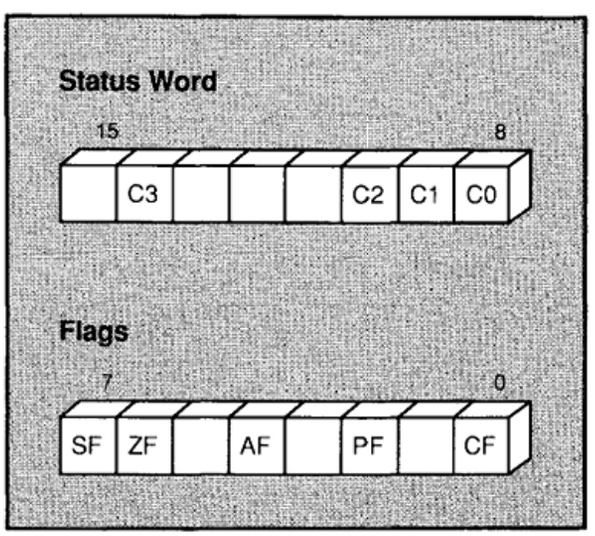


Figure 19.3 Coprocessor and Processor Control Flags

See Section 17.1.2 for more information on using conditional-jump instructions based on flag status.

19.7.1 Comparing Operands to Control Program Flow

The 8087-family coprocessors provide several instructions for comparing operands. All these instructions compare the stack top (**ST**) to a source operand, which may either be specified or implied as **ST(1)**.

The compare instructions affect the **C3**, **C2**, and **C0** control flags. The **C1** flag is not affected. Table 19.2 above shows the flags set for each possible result of a comparison or test.

Table 19.2

Control-Flag Settings after Compare or Test

After FCOM	After FTEST	C3	C2	C0
ST > *source*	**ST** is positive	0	0	0
ST < *source*	**ST** is negative	0	0	1
ST = *source*	**ST** is 0	1	0	0
Not comparable	**ST** is NAN or projective infinity	1	1	1

Variations on the compare instructions allow you to pop the stack once or twice, and to compare integers and zero. For each instruction, the stack top is always the implied destination operand. If you do not give an operand, **ST(1)** is the implied source. Some compare instructions allow you to specify the source as a memory or register operand.

The compare instructions are listed below.

Compare

These instructions compare the stack top to the source. The source and destination are unaffected by the comparison.

Syntax	Description
FCOM	Compares **ST** to **ST(1)**.
FCOM ST(*num***)**	Compares **ST** to **ST(***num***)**.
FCOM *mem*	Compares **ST** to *mem*. The memory operand can be a four- or eight-byte real number.

FICOM *mem*	Compares **ST** to *mem*. The memory operand can be a two- or four-byte integer.
FTST	Compares the **ST** to 0. The control registers will be affected as if **ST** had been compared to 0 in **ST(1)**. Table 19.2 above shows the possible results.

Compare and Pop

These instructions compare the stack top to the source, and then pop the stack. Thus the destination is destroyed by the comparison.

Syntax	Description
FCOMP	Compares **ST** to **ST(1)** and pops **ST** off the register stack.
FCOMP ST(*num***)**	Compares **ST** to **ST(***num***)** and pops **ST** off the register stack.
FCOMP *mem*	Compares **ST** to *mem* and pops **ST** off the register stack. The operand can be a four- or eight-byte real number.
FICOMP *mem*	Compares **ST** to *mem* and pops **ST** off the register stack. The operand can be a two- or four-byte integer.
FCOMPP	Compares **ST** to **ST(1)**, and then pops the stack twice. Both the source and destination are destroyed by the comparison.

80387 Only

Unordered compare instructions are available with the 80387. The **FUCOM**, **FUCOMP**, and **FUCOMPP** instructions are like **FCOM**, **FCOMP**, and **FCOMPP** except that the unordered versions do not cause invalid operation exceptions if one of the operands is a quiet NAN (not a number). Exceptions and NANs are beyond the scope of this manual and are not explained here. See Intel coprocessor reference books for more information.

■ Example

```
            IFDEF   c287
            .287
            ENDIF
            .DATA
down        DD      10.35       ; Sides of a rectangle
across      DD      13.07
diameter    DD      12.93       ; Diameter of a circle
status      DW      ?
            .CODE
            .
            .
            .
; Get area of rectangle
            fld     across      ; Load one side
            fmul    down        ; Multiply by the other

; Get area of circle
            fld1                ; Load one and
            fadd    st,st       ;   double it to get constant 2
            fdivr   diameter    ; Divide diameter to get radius
            fmul    st,st       ; Square radius
            fldpi               ; Load pi
            fmul                ; Multiply it

; Compare area of circle and rectangle
            fcompp              ; Compare and throw both away
            IFNDEF  c287
            fstsw   status      ; Load from coprocessor to memory
            fwait               ; Wait for coprocessor
            mov     ax,status   ; Memory to register
            ELSE
            fstsw   ax          ;   (for 287+, skip memory)
            ENDIF
            sahf                ;   to flags
            jp      nocomp      ; If parity set, can't compare
            jz      same        ; If zero set, they're the same
            jc      rectangle   ; If carry set, rectangle is bigger
            jmp     circle      ;  else circle is bigger

nocomp:     .                   ; Error handler
            .
same:       .                   ; Both equal
            .
rectangle:  .                   ; Rectangle bigger
            .
circle:     .                   ; Circle bigger
```

Notice how conditional blocks are used to enhance 80287 code. If you define the symbol `c287` from the command line by using the **/D***symbol* option (see Section 2.4.4, "Defining Assembler Symbols") the code is smaller and faster, but does not run on an 8087.

19.7.2 Testing Control Flags after Other Instructions

In addition to the compare instructions, the **FXAM** and **FPREM** instructions affect coprocessor control flags.

The **FXAM** instruction sets the value of the control flags based on the type of the number in the stack top (**ST**). This instruction is used to identify and handle special values such as infinity, zero, unnormal numbers, denormal numbers, and NANs (not a number). Certain math operations are capable of producing these special-format numbers. A description of them is beyond the scope of this manual. The possible settings of the flags are shown in the *Microsoft Macro Assembler Reference.*

FPREM also sets control flags. Since this instruction must sometimes be repeated to get a correct remainder for large operands, it uses the **C2** flag to indicate whether the remainder returned is partial (**C2** is set) or complete (**C2** is clear). If the bit is set, the operation should be repeated.

FPREM also returns the least-significant three bits of the quotient in **C0**, **C3**, and **C1**. These bits are useful for reducing operands of periodic transcendental functions, such as sine and cosine, to an acceptable range. The technique is not explained here. The possible settings for each flag are shown in the *Microsoft Macro Assembler Reference.*

19.8 Using Transcendental Instructions

The 8087-family coprocessors provide a variety of instructions for doing transcendental calculations, including exponentiation, logarithmic calculations, and some trigonometric functions.

Use of these advanced instructions is beyond the scope of this manual. However, the instructions are listed below for reference. All transcendental instructions have implied operands—either **ST** as a single destination operand, or **ST** as the destination and **ST(1)** as the source.

Instruction	Description
F2XM1	Calculates 2^x-1, where x is the value of the stack top. The value x must be between 0 and .5, inclusive. Returning 2^x-1 instead of 2^x allows the instruction to return the value with greater accuracy. The programmer can adjust the result to get 2^x.
FYL2X	Calculates Y times $\log_2 X$, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. The value of X must be positive.

FYL2XP1	Calculates Y times $\log_2 (X+1)$, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. The absolute value of X must be between 0 and the square root of 2 divided by 2. This instruction is more accurate than **FYL2X** when computing the log of a number close to 1.
FPTAN	Calculates the tangent of the value in **ST**. The result is a ratio Y/X, with Y replacing the value in **ST** and X pushed onto the stack so that after the instruction, **ST** contains Y and **ST(1)** contains X. The value being calculated must be a positive number less than pi/4. The result of the **FPTAN** instruction can be used to calculate other trigonometric functions, including sine and cosine.
FPATAN	Calculates the arctangent of the ratio Y/X, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. Both X and Y must be positive numbers less than infinity, and Y must be less than X. The result of the **FPATAN** instruction can be used to calculate other inverse trigonometric functions, including arcsine and arccosine.

■ 80387 Only

The following additional trigonometric functions are available on the 80387.

Instruction	Description
FSIN	Calculates the sine of the value in **ST**. The stack-top value is replaced by its sine.
FCOS	Calculates the cosine of the value in **ST**. The stack-top value is replaced by its cosine.
FSINCOS	Calculates the sine and cosine of the value in **ST**. When the instruction is complete, the value in **ST** is the cosine of the original stack-top value. The value in **ST(1)** is the sine of the original stack-top value. One of the values is pushed so that the former value in **ST(1)** is in **ST(2)**.

19.9 Controlling the Coprocessor

Additional instructions are available for controlling various aspects of the coprocessor. With the exception of **FINIT**, these instructions are generally used only by systems programmers. They are summarized below, but not fully explained or illustrated. Some instructions have a wait version and a no-wait version. The no-wait versions have **N** as the second letter.

Syntax	Description
F[N]INIT	Resets the coprocessor and restores all the default conditions in the control and status words. It is a good idea to use this instruction at the start and end of your program. Placing it at the start ensures that no register values from previous programs affect your program. Placing it at the end ensures that register values from your program will not affect later programs.
F[N]CLEX	Clears all exception flags and the busy flag of the status word. It also clears the error-status flag on the 80287 and 80387, or the interrupt-request flag on the 8087.
FINCSTP	Adds one to the stack pointer in the status word. Do not use to pop the register stack. No tags or registers are altered.
FDECSTP	Subtracts one from the stack pointer in the status word. No tags or registers are altered.
FREE ST(*num*)	Marks the specified register as empty.
FNOP	Copies the stack top to itself, thus padding the executable file and taking up processing time without having any effect on registers or memory.

■ 8087 Only

The 8087 has the instructions **FDISI**, **FNDISI**, **FENI**, and **FNENI**. These instructions can be used to enable or disable interrupts. The 80287 and 80387 coprocessors permit these instructions, but ignore them. Applications programmers will not normally need these instructions. Systems programmers should avoid using them so that their programs are portable to all coprocessors.

■ 80287/387 Only

Starting with the 80287, the **FSETPM** (Set Protected Mode) instruction is available. This instruction enables the coprocessor to run in protected mode. The primary difference is that the addresses stored in the instruction and operand pointers have a segment selector instead of an actual segment address. See Section 13.2, "Segmented Addresses," for information on segment selectors.

Either the **.286P** or **.386P** directive must be given before the **FSETPM** instruction can be used. Protected-mode operating systems normally set protected mode automatically. Therefore, you need this instruction only if you are writing control software.

CHAPTER 20

CONTROLLING THE PROCESSOR

20.1 Controlling Timing and Alignment 411
20.2 Controlling the Processor 411
20.3 Controlling Protected-Mode Processes 412
20.4 Controlling the 80386 413

The 8086-family processors provide instructions for processor control. Some of these instructions are available on all processors; others are for controlling protected-mode operations on the 80286 and 80386.

System-control instructions have limited use in applications programming. They are primarily used by systems programmers who write operating systems and other control software. Since systems programming is beyond the scope of this manual, the systems-control instructions are summarized, but not explained in detail, in the sections below.

20.1 Controlling Timing and Alignment

The **NOP** instruction does nothing but take up time and space. It works by exchanging the **AX** with itself. The **NOP** instruction can be used for delays in timing loops, or to pad executable code for alignment.

Normally, applications programmers should avoid using the **NOP** instruction in timing loops, since such loops take different lengths of time on different machines. A better way to control timing is to use the DOS time function, since it is based on the computer's internal clock rather than on the speed of the processor.

MASM automatically inserts **NOP** instructions for padding when you use the **ALIGN** or **EVEN** directive (see Section 6.5, "Aligning Data") to align data or code on a given boundary. The assembler automatically inserts **NOP** instructions for alignment.

20.2 Controlling the Processor

The **WAIT**, **ESC**, **LOCK**, and **HLT** instructions control different aspects of the processor.

These instructions can be used to control processes handled by external coprocessors. The 8087-family coprocessors are the coprocessors most

commonly used with 8086-family processors, but 8086-based machines can work with other coprocessors if they have the proper hardware and control software.

These instructions are summarized below:

Instruction	Description
LOCK	Locks out other processors until a specified instruction is finished. This is a prefix that precedes the instruction. It can be used to make sure that a coprocessor does not change data being worked on by the processor.
WAIT	Instructs the processor to do nothing until it receives a signal that a coprocessor has finished with a task being performed at the same time. See Section 19.4, "Coordinating Memory Access," for information on using **WAIT** or its coprocessor equivalent, **FWAIT**, with the 8087-family coprocessors.
ESC	Provides an instruction and possibly a memory operand for use by a coprocessor. **MASM** automatically inserts **ESC** instructions when required for use with 8087-family coprocessors.
HLT	Stops the processor until an interrupt is received. It can be used in place of an endless loop if a program needs to wait for an interrupt.

20.3 Controlling Protected-Mode Processes

■ 80286/386 Only

Protected mode is available starting with the 80286 processors. This mode is generally initiated and controlled by an operating system. Current versions of DOS do not support protected mode.

The instructions that control protected mode are privileged and can only be used if the **.286P** or **.386P** directives have been given. These instructions are generally needed only for operating systems and other control software.

Note that, under protected-mode operating systems such as XENIX and OS/2, applications programmers do not need to use protected-mode instructions. Process control is managed through system calls.

Some privileged-mode instructions use internal registers of the 80286 or 80386 processors. Instructions are provided for loading values from these registers into memory where the values can be modified. Other instructions can then be used to store the values back to the special registers.

The privileged-mode instructions are listed below:

Instruction	Description
LAR	Loads access rights
LSL	Loads segment limit
LGDT	Loads global descriptor table
SGDT	Stores global descriptor table
LIDT	Loads 8-byte-interrupt descriptor table
SIDT	Stores 8-byte-interrupt descriptor table
LLDT	Loads local descriptor table
SLDT	Stores local descriptor table
LTR	Loads task register
STR	Stores task register
LMSW	Loads machine-status word
SMCW	Stores machine-status word
ARPL	Adjusts requested privilege level
CLTS	Clears task-switched flag
VERR	Verifies read access
VERW	Verifies write access

20.4 Controlling the 80386

■ 80386 Only

The 80386 processor can use all the privileged-mode instructions of the 80286, but it also allows you to use **MOV** to transfer data between general-purpose registers and special registers.

The following special registers can be accessed with move instructions on the 80386:

Type	Registers
Control	**CR0**, **CR2**, and **CR3**
Debug	**DR0**, **DR1**, **DR2**, **DR3**, **DR6**, and **DR7**
Test	**TR6** and **TR7**

These registers can be moved directly to 32-bit registers or from them.

■ Examples

```
mov     eax,cr0             ; Load CR0 into EAX
mov     cr1,ecx             ; Store ECX in CR1
```

APPENDIXES

A New Features 417
B Error Messages and Exit Codes 429

APPENDIX A
NEW FEATURES

A.1 MASM Enhancements 419
A.1.1 80386 Support 419
A.1.2 Segment Simplification 420
A.1.3 Performance Improvements 420
A.1.4 Enhanced Error Handling 421
A.1.5 New Options 421
A.1.6 Environment Variables 422
A.1.7 String Equates 422
A.1.8 RETF and RETN Instructions 422
A.1.9 Communal Variables 422
A.1.10 Including Library Files 422
A.1.11 Flexible Structure Definitions 423
A.2 Link Enhancements 423
A.3 The CodeView Debugger 423
A.4 SETENV 424
A.5 Compatibility with Assemblers and Compilers 424

Version 5.0 of the Microsoft Macro Assembler package has many significant new features. Some of the most important are the new CodeView debugger, the support for the 80386 processor, and an optional simplified system of defining segments. This appendix describes these features and tells you where they are documented.

A.1 MASM Enhancements

MASM, the Macro Assembler program, now has several important enhancements over Version 4.0 and other previous versions. The sections below summarize new options, directives, instructions, and other features.

A.1.1 80386 Support

MASM now supports the 80386 instruction set and addressing modes. The 80386 processor is a superset of other 8086-family processors. Most new features are simply 32-bit extensions of 16-bit features.

If you understand the features of the 16-bit 8086-family processors, then using the 32-bit extensions is not difficult. The new 32-bit registers are used in much the same way as the 16-bit registers. The 80386 registers are explained in Section 13.3.

However, some features of the 80386 processor are significantly different. Throughout the manual the heading "80386 Only" is used to flag sections in which 80386 enhancements are described. Areas of particular importance include the **.386** directive for initializing the 80386 (Section 4.4.1, "Defining Default Assembly Behavior"), the **USE32** and **USE16** segment types for setting the segment word size (Section 5.2.2.2), and indirect addressing modes (Section 14.3.3, "80386 Indirect Memory Operands").

The 80386 processor and the 80387 coprocessor also have the new instructions listed in Table A.1 below.

Table A.1

80386 and 80387 Instructions

Name	Mnemonic	Reference
Bit Scan Forward	**BSF**	Section 16.7
Bit Scan Reverse	**BSR**	Section 16.7
Bit Test	**BT**	Section 17.1.2.4
Bit Test and Complement	**BTC**	Section 17.1.2.4
Bit Test and Reset	**BTR**	Section 17.1.2.4
Bit Test and Set	**BTS**	Section 17.1.2.4
Move with Sign Extend	**MOVSX**	Section 15.2.3
Move with Zero Extend	**MOVZX**	Section 15.2.3
Set Byte on Condition	**SET***condition*	Section 17.3
Double Precision Shift Left	**SHLD**	Section 16.8.5
Double Precision Shift Right	**SHRD**	Section 16.8.5
Move to/from Special Registers	**MOV**	Section 18.1
Sine	**FSIN**	Section 19.8
Cosine	**FCOS**	Section 19.8
Sine Cosine	**FSINCOS**	Section 19.8
IEEE Partial Remainder	**FPREM1**	Section 19.6
Unordered Compare Real	**FUCOM**	Section 19.7.1
Unordered Compare Real and Pop	**FUCOMP**	Section 19.7.1
Unordered Compare Real and Pop Twice	**FUCOMPP**	Section 19.7.1

A.1.2 Segment Simplification

A new system of defining segments is available in **MASM** Version 5.0. The simplified segment directives use the Microsoft naming conventions. If you are willing to accept these conventions, segments can be defined more easily and consistently. However, this feature is optional. You can still use the old system if you need more direct control over segments or if you need to be consistent with existing code. See Section 5.1, "Simplified Segment Definitions."

A new **DOSSEG** directive enables you to specify DOS segment order in the source file. This directive is equivalent to the **/DOSSEG** option of the linker. See Section 5.1.2, "Specifying DOS Segment Order."

A.1.3 Performance Improvements

The performance of **MASM** has been enhanced in two ways: faster assembly and larger symbol space.

Version 5.0 of the assembler is significantly faster for most source files. The improvement varies depending on the relative amounts of code and data in the source file, and on the complexity of expressions used.

Symbol space is now limited only by the amount of system memory available to your machine.

A.1.4 Enhanced Error Handling

Error handling has been enhanced in the following ways:

- Messages have been reworded, enhanced, or reorganized.
- Messages are divided into three levels: severe errors, serious warnings, and advisory warnings. The level of warning can be changed with the **/W** option. Type-checking errors are now serious warnings rather than severe errors. See Section 2.4.13, "Setting the Warning Level."
- During assembly, messages are output to the standard output device (by default, the screen). They can be redirected to a file or device. In Version 4.0 they were sent to the standard error device. See Section 2.3, "Controlling Message Output."

A.1.5 New Options

The following command-line options have been added:

Option	Description
/W[[**0**|**1**|**2**]]	Sets the warning level to determine what type of messages will be displayed. The three kinds are severe errors, serious warnings, and advisory warnings. See Section 2.4.13, "Setting the Warning Level."
/ZI and **/ZD**	Sends debugging information for symbolic debuggers to the object file. The **/ZD** option outputs line-number information, whereas the **/ZI** option outputs both line-number and type information. See Section 2.4.16, "Writing Symbolic Information to the Object File."
/H	Displays the **MASM** command line and options. See Section 2.4.5, "Creating Code for a Floating-Point Emulator."
/D*sym*[[=*val*]]	Allows definition of a symbol from the command line. This is an enhancement of a current option. See Section 2.4.4, "Defining Assembler Symbols" in Part 1.

In addition, the new directives **.ALPHA** and **.SEQ** have been added; these directives have the same effect as the **/A** and **/S** options. See Section 5.2.1, "Setting the Segment-Order Method."

A.1.6 Environment Variables

MASM now supports two environment variables: **MASM** for specifying default options, and **INCLUDE** for specifying the search path for include files. See Section 2.2, "Using Environment Variables."

A.1.7 String Equates

String equates have been enhanced for easier use. By enclosing the argument to the **EQU** directive in angle brackets, you can ensure that the argument is evaluated as a string equate rather than as an expression. See Section 11.1.3, "Using String Equates."

The expression operator (%) can now be used with macro arguments that are text macros as well as arguments that are expressions. See Section 11.4.4, "Using the Expression Operator."

A.1.8 RETF and RETN Instructions

The **RETF** (Return Far) and **RETN** (Return Near) instructions are now available. These instructions enable you to define procedures without the **PROC** and **ENDP** directives. See Section 17.4.2, "Defining Procedures."

A.1.9 Communal Variables

MASM now allows you to declare communal variables. These uninitialized global data items can be used in include files. They are compatible with variables declared in C include files. See Section 8.3, "Using Multiple Modules."

A.1.10 Including Library Files

The **INCLUDELIB** directive enables you to specify in the assembly source file any libraries that you want to be linked with your program modules. See Section 8.5, "Specifying Library Routines."

A.1.11 Flexible Structure Definitions

Structure definitions can now include conditional-assembly statements, thus enabling more flexible structures. See Section 7.1.1, "Declaring Structure Types."

A.2 Link Enhancements

LINK has several new features. These enhancements are discussed in Chapter 12, "Linking Object Files with LINK," of the Microsoft CodeView and Utilities manual. They are summarized below:

- The **LINK** environment specifies default linker options.
- The **TMP** environment variable specifies a directory in which **link** can create temporary files if it runs out of memory.
- The **/CODEVIEW** option puts debugging information in executable files for the CodeView debugger.
- The **/INFORMATION** option displays each step of the linking process including parsing the command line, Pass 1, and so on. The path and name of each module are displayed as the modules are linked.
- The **/BATCH** option disables the linker's prompting interface so that make or batch files are not be stopped by **link** errors.
- The **/QUICKLIB** option creates a user's library for a Microsoft Quick language (such as QuickBASIC).
- The **/FARCALLTRANSLATION** and **/PACKCODE** options enable two optimizations that may make code faster in certain situations.

A.3 The CodeView Debugger

In Version 5.0 of the Macro Assembler package, the CodeView debugger replaces **SYMDEB**. This source-level symbolic debugger is capable of working with programs developed with **MASM** or with Microsoft high-level-language compilers.

The CodeView debugger features a window-oriented environment with multiple windows displaying different types of information. Commands can be executed with a mouse, function keys, or command lines. Variables can be watched in a separate window as the program executes.

MASM and **LINK** have been enhanced to support the features of the CodeView debugger.

A.4 SETENV

Since **MASM** and **LINK** now support more environment variables, users may wish to define environment strings that exceed the default size of the DOS environment. The **SETENV** program in the CodeView and Utilities manual is provided as a means of modifying the environment size for DOS Versions 2.0 to 3.1.

A.5 Compatibility with Assemblers and Compilers

If you are upgrading from a previous version of the Microsoft or IBM Macro Assembler, you may need to make some adjustments before assembling source code developed with previous versions. The potential compatibility problems are listed below:

- All previous versions of the Macro Assembler assembled initialized real-number variables in the Microsoft Binary format by default. Version 5.0 assembles initialized real-number variables in the IEEE format. If you have source modules that depend on the default format being Microsoft Binary, you must modify them by placing the **.MSFLOAT** directive at the start of the module before the first variable is initialized.

  In previous versions of the Macro Assembler, the default conditions were 8086 instructions enabled, coprocessor instructions disabled, and real numbers assembled in Microsoft Binary format. The **/R** option, the **.8087** directive, or the **.287** directive was required to enable coprocessor instructions and IEEE format for real numbers. In Version 5.0, the default conditions are 8086 and 8087 instructions enabled and real numbers assembled in IEEE format. Although the **/R** option is no longer used, it is recognized and ignored so that existing make and batch files work without modification.

- Some previous versions of the IBM Macro Assembler wrote segments to object files in alphabetical order. The current version writes segments to object files in the order encountered in the source file. You can use the **/A** option or the **.ALPHA** directive to order segments alphabetically if this segment order is required for

your existing source code. See Section 5.2.1, "Setting the Segment-Order Method," for more information.

- Some early versions of the Macro Assembler did not have strict type checking. Later versions had strict type checking that produced errors on source code that would have run under the earlier versions. **MASM** Version 5.0 solves this incompatibility by making type errors into warning messages. You can set the warning level so that type warnings will not be displayed, or you can modify the code so that the type is given specifically. Section 9.5, "Strong Typing for Memory Operands," describes strict type checking and how to modify source code developed without this feature.

The programs in the Microsoft Macro Assembler package are compatible with Microsoft (and most IBM) high-level languages. An exception occurs when the current version of **LINK** is used with IBM COBOL 1.0, IBM FORTRAN 2.0, or IBM Pascal 2.0. If source code developed with these compilers has overlays, you must use the linker provided with the compiler. Do not use the new version of **LINK** provided with the assembler.

APPENDIX B
ERROR MESSAGES AND EXIT CODES

B.1 MASM Messages and Exit Codes.....429
B.1.1 Assembler Status Messages 429
B.1.2 Numbered Assembler Messages.....430
B.1.3 Unnumbered Error Messages 446
B.1.4 MASM Exit Codes.....448
B.2 CREF Error Messages and Exit Codes 449

This appendix lists and explains the messages and exit codes that can be generated by **MASM** and **CREF**.

Messages are sent to the standard output device. By default, this device is the screen, but you can redirect the messages to a file or to a device such as a printer.

B.1 MASM Messages and Exit Codes

The assembler can display several kinds of messages as well as output an exit code; the kind of exit code output depends on the error, if any, encountered during the assembly.

B.1.1 Assembler Status Messages

After every assembly, **MASM** reports on the symbol space, errors, and warnings. A sample display is shown below:

```
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981, 1987.  All rights reserved.


  47904 + 353887 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors
```

The first line indicates how much near and far symbol space was unused during the assembly. This data may help you determine whether increasing the size of your program will exhaust available memory.

The first number indicates near symbol space. There is 64K total. The second number indicates far symbol space. This is equal to the size of **MASM**, the size of **MASM** buffers, and the amount of available memory less near data space. Most symbols go into far space. When far space is exhausted, additional symbols go into near space. Using both far and near space causes a decrease in speed of assembly.

You can use the **/V** option to direct **MASM** to display the addition statistics. The number of source lines, the total number of source- and include-file lines, and the number of symbols are shown. This information appears only if no severe errors are encountered. An example is shown below:

```
    742 Source  Lines
    799 Total   Lines
     44 Symbols
```

The /T option can be used to suppress all output to the screen after assembly.

B.1.2 Numbered Assembler Messages

The assembler displays messages on the screen whenever it encounters an error while processing a source file. It also displays a warning message whenever it encounters questionable syntax. Messages that can be associated with a particular line of code are numbered. General errors related to the entire assembly rather than to a particular line are unnumbered (see Section B.1.3).

Numbered error messages are displayed in the following format:

sourcefile(*line*) : *code*: *message*

The *sourcefile* is the name of the source file where the error occurred. If the error occurred in a macro in an include file, the *sourcefile* is the file where the macro was called and expanded—not the file where it was defined.

The *line* indicates the point in the source file where **MASM** was no longer able to assemble.

The *code* is an identifying code in the format used by all Microsoft language programs. It starts with the word "error" or "warning" followed by a five-character code. The first character is a letter indicating the program or language. Assembler messages start with A. The first digit indicates the warning level. The number is 2 for severe errors, 4 for serious warnings, and 5 for advisory warnings. The next three digits are the error number. For example, severe error 38 is shown as A2038.

The *message* is a descriptive line describing the error.

MASM messages are listed in numerical order in this section with a short explanation for each.

Note

Some numbers in sequence are not assigned messages because errors that could be generated in previous versions of **MASM** have been removed or reorganized in this version.

Code	Message
0	`Block nesting error` Nested procedures, segments, structures, macros, or repeat blocks were not properly terminated. This error may indicate that you closed an outer level of nesting with inner levels still open.
1	`Extra characters on line` Sufficient information to define a statement has been received on a line, but additional characters were also provided. This may indicate that you provided too many arguments.
2	`Internal error - Register already defined` *symbol* Note the conditions when the error occurs and contact Microsoft Corporation by using the Product Assistance Request form at the end of this manual.
3	`Unknown type specifier` An invalid type specifier was used to give the size of a label or external declaration. For instance, **BYTE** or **NEAR** might have been misspelled.
4	`Redefinition of symbol` A symbol was defined in two places with different types. This error occurs during Pass 1 on the second declaration of the symbol.
5	`Symbol is multidefined:` A symbol is defined in two places. This error occurs during Pass 2 on each declaration of the symbol.
6	`Phase error between passes` An ambiguous instruction or directive caused the relative address of a label to be changed between Pass 1 and Pass 2. You can use the **/D** option to produce a Pass 1 listing to aid in resolving phase errors between passes. The format of Pass 1 listings is discussed in Section 2.5.7.
7	`Already had ELSE clause` More than one **ELSE** clause was used within a conditional assembly block. Each nested **ELSE** must have its own **IF** directive and **ENDIF**.

8 `Must be in conditional block`

An **ENDIF** or **ELSE** was specified without a corresponding **IF** directive.

9 `Symbol not defined:`

A symbol was used without being defined. This error is produced for forward references on the first pass and is ignored if the references are resolved on the second pass.

10 `Syntax error`

A statement did not match any recognizable assembler syntax. **MASM** tries to be specific, so this error only occurs if the statement bears no resemblance to any legal statement.

11 `Type illegal in context`

The type specifier was given with an unacceptable size. For example, a procedure was defined as having **BYTE** type, instead of **NEAR** or **FAR** type.

12 `Group name must be unique`

A name assigned as a group name was already defined as another type of symbol.

13 `Must be declared during Pass 1:` *symbol*

An item was referenced before it was defined in Pass 1. For example, `IF DEBUG` is illegal if the symbol `DEBUG` was not previously defined.

14 `Illegal public declaration`

A symbol was declared public illegally. For instance, a text equate cannot be declared public. Section 8.1, "Declaring Symbols Public," explains public declarations.

15 `Symbol already different kind:` *symbol*

A symbol was redefined to a different kind of symbol. For example, a segment name was reused as a variable name, or a structure name was reused as an equate name.

16 `Reserved word used as symbol:` *name*

An assembler keyword was used as a symbol. This is a warning, not an error, and can be ignored if you wish. However, the keyword is no longer available for its original purpose. For example, if you name a macro `add`, it replaces the **ADD** instruction.

17 `Forward reference illegal`

A symbol was referenced before it was defined on Pass 1. For example, the following lines produce an error:

```
        DB      count DUP(?)
count   EQU     10
```

The statements would be legal if the lines were reversed.

18 `Operand must be register:` *operand*

A register was expected as an operand, but a symbol or constant was supplied.

20 `Operand must be segment or group`

A segment or group name was expected, but some other kind of operand was given. For instance, the **ASSUME** directive requires that the symbol assigned to a segment register be a segment name, a group name, a **SEG** expression, or a text equate representing a segment or group name. Thus the following statement is accepted:

```
ASSUME  ds:SEG variable  ; Legal
```

However, if the same statement is assigned to an equate, it is not accepted, as shown below:

```
segvar      EQU     SEG variable
            ASSUME  ds:segvar       ; Illegal
```

22 `Operand must be type specifier`

An operand was expected to be a type specifier, such as **NEAR** or **FAR**, but some other kind of operand was received.

23 `Symbol already defined locally`

A symbol that had already been defined within the current module was declared **EXTRN**.

24 `Segment parameters are changed`

A segment declaration with the same name as a previous segment declaration was given with arguments that did not match the previous declaration. See Section 5.2 for information on defining segments.

25 `Improper align/combine type`

SEGMENT parameters are incorrect. Check the align and combine types to make sure you have entered valid types

from among those discussed in Section 5.2, "Full Segment Definitions."

26 `Reference to multidefined symbol`

An instruction referenced a symbol defined in more than one place.

27 `Operand expected`

An operand was expected, but an operator was received.

28 `Operator expected`

An operator was expected, but an operand was received.

29 `Division by 0 or overflow`

An expression resulted in division by 0 or in a number too large to be represented.

30 `Negative shift count`

An expression using the **SHR** or **SHL** operator evaluated to a negative shift count.

31 `Operand types must match`

An instruction received operands of different sizes. For example, this warning is generated by the following code:

```
string      DB      "This is a test"
            .
            .
            .
            mov     ax,string[4]
```

Since this is a warning rather than an error, **MASM** attempts to generate code based on its best guess of the intended result. If one of the operands is a register, the register size overrides the size of the other operand. In the example, the word size of **AX** overrides the byte size of `string[4]`. You can avoid this warning and make your code less ambiguous by specifying the operand size with the **PTR** operator. For example:

```
            move    ax,WORD PTR string[4]
```

32 `Illegal use of external`

An external variable was used incorrectly. See Section 8.2 for information about correct declaration and use of external symbols.

34 `Operand must be record or field name`

An operand was expected to be a record name or record-field name, but another kind of operand was received.

35 `Operand must have size`

An operand was expected to have a specified size, but no size was supplied. For example, the following statement is illegal:

```
inc     [bx]
```

Often this error can be remedied by using the **PTR** operator to specify a size type, as shown below:

```
inc     BYTE PTR [bx]
```

38 `Left operand must have segment`

The left operand of a segment-override expression must be a segment register, group, or segment name. For example, if `mem1` and `mem2` are variables, the following statement is illegal:

```
mov     dx,mem1:mem2
```

39 `One operand must be constant`

The addition operator was used incorrectly. For instance, two memory operands cannot be added in an expression. Valid uses of the addition operator are explained in Section 9.2.1.1, "Arithmetic Operators."

40 `Operands must be in same segment, or one must be constant`

The subtraction operator was used incorrectly. For instance, a memory operand in the code segment cannot be subtracted from a memory operand in the data segment. Valid uses of the subtraction operator are explained in Section 9.2.1.1.

42 `Constant expected`

A constant operand was expected, but an operand or expression that does not evaluate to a constant was supplied.

43 `Operand must have segment`

The **SEG** operator was used incorrectly. For instance, a constant operand cannot have a segment. See Section 9.2.4.5 for a description of valid uses of the **SEG** operator.

44 Must be associated with data

A code-related item was used where a data-related item was expected.

45 Must be associated with code

A data-related item was used where a code-related item was expected.

46 Multiple base registers

More than one base register was used in an operand. For example, the following line is illegal:

```
mov     ax,[bx+bp]
```

47 Multiple index registers

More than one index register was used in an operand. For example, the following line is illegal:

```
mov     ax,[si+di]
```

48 Must be index or base register

An indirect memory operand requires a base or index register, but some other register was specified. For example, the following line is illegal:

```
mov     ax,[bx+ax]
```

Only **BP**, **BX**, **DI**, and **SI** may be used in indirect operands (except with 32-bit registers on the 80386).

49 Illegal use of register

A register was used in an illegal context. For example, the following statement is illegal:

```
mov     ax,cs:[si]
```

50 Value out of range

A value was too large for its context. For example,

```
mov     al,5000
```

is illegal; you must use a byte value for a byte register.

51 Operand not in current CS ASSUME segment

An operand was used to represent a code address outside the code segment assigned with the **ASSUME** statement. This usually indicates a call or jump to a label outside the

current code segment.

52 `Improper operand type: symbol`

An illegal operand was given for a particular context. For example

```
mov     mem1,mem2
```

is illegal if both operands are memory operands.

53 `Jump out of range by number bytes`

A conditional jump was not within the required range. For all except the 80386 processor, the range is 128 bytes backward or 127 bytes forward from the start of the instruction following the jump instruction. For the 80386, the default range is from –32,768 to 32,767. You can usually correct the problem by reversing the condition of the conditional jump and using an unconditional jump (**JMP**) to the out-of-range label, as described in Section 9.4.1, "Forward References to Labels."

55 `Illegal register value`

A register was specified with an illegal syntax. For example, you cannot access a stack variable with the following:

```
mov     ax,bp+4
```

The correct syntax (as explained in Section 17.4.3, "Passing Arguments on the Stack") is shown below:

```
mov     ax,[bp+4]
```

56 `Immediate mode illegal`

An immediate operand was supplied to an instruction that cannot use immediate data. For example, the following statement is illegal:

```
mov     ds,DGROUP
```

You must move the segment address into a general register and then move it from that register to **DS**.

57 `Illegal size for operand`

The size of an operand is illegal with the specified instruction. For instance, you cannot use a shift or rotate instruction with a doubleword (except on the 80386). Since this is a warning rather than an error, **MASM** does assemble code for the instruction, making a reasonable guess at your intention. For example, if the statement

```
inc     mem32
```

is given where `mem32` is a doubleword memory operand, **MASM** actually only increments the low-order word of the operand, since a word is the largest operand that can be incremented (except on the 80386). This error may occur if you try to assemble source code written for assemblers that have less strict type checking than the Microsoft Macro Assembler (such as early versions of the IBM Macro Assembler). Usually you can solve the problem by specifying the size of the item with the **PTR** operator, as explained in Section 9.5, "Strong Typing for Memory Operands."

58 `Byte register illegal`

A byte register was used in a context where a word register (or 32-bit register on the 80386) is required. For example, `push al` is illegal; use `push ax` instead.

59 `Illegal use of CS register`

The **CS** register was used in an illegal context, such as those listed below:

```
pop     cs
mov     cs,ax
```

60 `Must be accumulator register`

A register other than **AL**, **AX**, or **EAX** was supplied in a context where only the accumulator register is acceptable. For instance, the **IN** instruction requires the accumulator register as its left (destination) operand.

61 `Improper use of segment register`

A segment register was used in a context where it is illegal. For example, `inc cs` is illegal.

62 `Missing or unreachable code segment`

A jump was attempted to a label in a segment that **MASM** does not recognize as a code segment. This usually indicates that there is no **ASSUME** statement associating the **CS** register with a segment.

63 `Operand combination illegal`

Two operands were used with an instruction that does not allow the specified combination of operands. For example, the following operand combination is illegal:

```
xchg    mem1,mem2
```

64 `Near JMP/CALL to different code segment`

A near jump or call instruction attempted to access an address in a code segment other than the one used in the currently active **ASSUME**. To correct the error, use a far call or jump, or use an **ASSUME** statement to change the code segment currently referenced by **CS**. See Section 5.4, "Associating Segments with Registers," for information on the **ASSUME** directive.

65 `Label cannot have segment override`

A segment override was used incorrectly. See Section 9.2.3 for examples of valid uses of the segment override operator.

66 `Must have instruction after prefix`

A repeat prefix such as **REP**, **REPE**, or **REPNE** was given without specifying the instruction to repeat.

67 `Cannot override ES for destination`

A segment override was used on the destination of a string instruction. Although the default **DS:SI** register pair for the source can have a segment override, the destination must always be in the **ES:DI** register pair. The **ES** segment cannot be overridden. For example, the following statement is illegal:

```
rep     stos ds:destin      ; Can't override ES
```

68 `Cannot address with segment register`

A statement tried to access a memory operand, but no **ASSUME** directive had been used to specify a segment for the operand. See Section 5.4, "Associating Segments with Registers," for information on the **ASSUME** directive.

69 `Must be in segment block`

A directive (such as **EVEN**) that is expected to be in a segment is used outside a segment.

70 `Cannot use EVEN or ALIGN with byte alignment`

The **EVEN** or **ALIGN** directive was used in a segment that is byte aligned. Section 6.5, "Aligning Data," explains the **EVEN** and **ALIGN** directives.

71 `Forward reference needs override or FAR`

A call or jump attempts to access a far label that was not declared far earlier in the source code. You can use the **PTR** operator to specify far calls and jumps, as shown below:

```
call    FAR PTR task
jmp     FAR PTR location
```

72 `Illegal value for DUP count`

The count value specified for a **DUP** operator did not evaluate to a constant integer greater than 0.

73 `Symbol is already external`

A symbol that had already been declared external was later defined locally. Section 8.2 describes external declarations.

74 `DUP nesting too deep`

DUP operators were nested to more than 17 levels.

75 `Illegal use of undefined operand (?)`

The undefined operand (**?**) was used incorrectly. For example, the following statements are illegal:

```
stuff   DB      5 DUP (?+5) ; Can't use in expression
        mov     ax,?        ; Can't use in code
```

Valid uses of the undefined operand are explained in Section 6.3.2, "Defining Arrays and Buffers."

76 `Too many values for structure or record initialization`

Too many initial values were given when declaring a record or structure variable. The number of values in the declaration must match the number in the definition. For example, a structure `test` defined with four fields could be declared as shown below:

```
stest       test    <4,,'c',0>
```

The declaration must have four or fewer fields.

77 `Angle brackets required around initialized list`

A structure variable was defined without angle brackets around the initial values in the list. For example, the following definition is illegal:

```
stest       test    4,,'c'0
```

The following definitions are correct:

```
stest    test  <4,,'c',0> ; Three initial values, one blank
ttest    test  <>         ; No initial values
```

78 Directive illegal in structure

A statement within a structure definition was not one of the following: a data definition using define directives such as **DB** or **DW**, a comment preceded by a semicolon, or a conditional-assembly directive.

79 Override with DUP illegal

The **DUP** operator was used in a structure initialization list. For example, the following example is illegal because of the **DUP** operator:

```
stest       test    <3,4 DUP (3),5>
```

80 Field cannot be overridden

An item in a structure-initialization list attempted to override a structure field that could not be overridden. For instance, if a field is initialized in the structure definition with the **DUP** operator, it cannot be overridden in a declaration. See the note in Section 7.1.2, "Defining Structure Variables."

83 Circular chain of EQU aliases

An alias declared with the **EQU** directive points to itself. For example, the following lines are illegal:

```
a           EQU     b
b           EQU     a
```

84 Cannot emulate coprocessor opcode

Either a coprocessor instruction or operands used with such an instruction produced an opcode that the coprocessor emulator does not support. Since the emulator library is not supplied with the Microsoft Macro Assembler, this error can only occur if you are linking assembler routines with code from a high-level-language compiler that uses the emulator.

85 End of file, no END directive

The source code was not terminated by an **END** statement. This error can also occur as the result of segment-nesting errors.

86 `Data emitted with no segment`

A statement that generates code or data was used outside all segment blocks. Instructions and data declarations must be in segments, but directives that specify assembler behavior without generating code or data can be outside segments.

87 `Forced error - pass1`

An error was forced with the **.ERR1** directive.

88 `Forced error - pass2`

An error was forced with the **.ERR2** directive.

89 `Forced error`

An error was forced with the **.ERR** directive.

90 `Forced error - expression true (0)`

An error was forced with the **.ERRE** directive.

91 `Forced error - expression false (not 0)`

An error was forced with the **.ERRNZ** directive.

92 `Forced error - symbol not defined`

An error was forced with the **.ERRNDEF** directive.

93 `Forced error - symbol defined`

An error was forced with the **.ERRDEF** directive.

94 `Forced error - string blank`

An error was forced with the **.ERRB** directive.

95 `Forced error - string not blank`

An error was forced with the **.ERRNB** directive.

96 `Forced error - strings identical`

An error was forced with the **.ERRIDN** directive.

97 `Forced error - strings different`

An error was forced with the **.ERRDIF** directive.

98 `Wrong length for override value`

The override value for a structure field is too large to fit in the field. An example is shown below:

```
x           STRUC
x1          DB      "A"
x           ENDS

y           x       <"AB">
```

The override value is a string consisting of two bytes; the structure declaration provided only room for one byte.

99 `Line too long expanding symbol: symbol`

An equate defined with the **EQU** directive was so long that expanding it caused the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

100 `Impure memory reference`

Data was stored into the code segment when the **/P** option and privileged instructions (enabled with **.286P** or **.386P**) were in effect. An example of storing data in the code segment is shown below:

```
            .CODE
c_word      DW      ?
            .
            .
            .
            mov     cs:c_word,data
```

The **/P** option checks for such statements, which are acceptable in real mode, but can cause problems in privileged mode.

101 `Missing data; zero assumed`

An operand is missing from a statement, as shown below:

```
        mov     ax,
```

Since some programmers use this syntax purposely, the message is a warning. **MASM** assumes that 0 was intended and assembles the following code:

```
        mov     ax,0
```

102 `Segment near (or at) 64K limit`

A bug in the 80286 processor causes jump errors when a code segment approaches within a few bytes of the 64K limit in privileged mode. This error warns about code that may fail because of the bug. The error can only be generated when the **.286** directive is given.

103 `Align must be power of 2`

A number that is not a power of two was used with the **ALIGN** directive. The directive is explained in Section 6.5, "Aligning Data."

104 `Jump within short distance`

A **JMP** instruction was used to jump to a short label (128 or fewer bytes before the end of the **JMP** instruction, or 127 or fewer bytes beyond the instruction). By default the assembler assumes that jumps are near (greater than short, but still in one segment). If a short jump is encountered, **MASM** uses a short form of the **JMP** instruction (2 bytes) rather than the long form (3 bytes with 16-bit segments or 5 bytes with 32-bit segments). You can make your code slightly more efficient by using the **SHORT** operator to specify that a jump is short rather than near. For example, using the **SHORT** operator in the following example saves 1 byte of code:

```
        jmp     SHORT there
        .
        .
there:  .                       ; Less than 127 bytes
```

Using the **SHORT** operator with forward references to code labels is explained in Section 9.2.4.2. With the 80386 processor, this message also applies to conditional jumps, which can be either short (2 bytes) or near (4 bytes).

105 `Expected` *element*

An element such as a punctuation mark or operator was omitted. For instance, if you omit the comma between source and destination operands, the message `Expected comma` is generated.

106 `Line too long`

A source line was longer than 128 characters, the maximum allowed by **MASM**.

107 `Illegal digit in number`

A constant number contained a digit that is not allowed in the current radix.

108 `Empty string not allowed`

A statement used an empty string. For example, the following definition is illegal:

```
null        DB      ""
```

In many languages an empty string represents ASCII character 0. In assembly language, you must give the value 0, as shown below:

```
null        DB      0
```

109 `Missing operand`

The instruction or directive requires more operands than were provided.

110 `Open parenthesis or bracket`

Only one parenthesis or bracket was given in a statement that requires opening and closing parentheses or brackets.

111 `Directive must be in macro`

A directive that is expected only in macro definitions was used outside a macro.

112 `Unexpected end of line`

A line ended before a complete statement was formed. **MASM** expects more information, but cannot identify what information is missing.

113 `Cannot change processor in segment`

A processor directive was encountered within a segment. Processor directives must be given before the first segment directive or between segments. If you want to change the processor in the middle of the segment, you must close the current segment, give the processor directive, and then start another segment.

114 `Operand size does not match segment word size`

A 32-bit operand was used in a 16-bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is a questionable practice in a 32-bit segment:

```
mov   ax,OFFSET nearlabel ; Load near (32-bit) label
```

The following statement is a questionable practice in a 16-bit segment:

```
mov   eax,OFFSET farlabel ; Load far (48-bit) label
```

This is a warning that you can ignore if you are certain you know what you are doing.

115 Address size does not match segment word size

A 32-bit address was used in a 16-bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is a questionable practice in a 32-bit segment:

```
mov   eax,[si]   ; Load value pointed to by 16-bit pointer
```

The following statement is a questionable practice in a 16-bit segment:

```
mov   ax,[esi]   ; Load value pointed to by 32-bit pointer
```

This is a warning that you can ignore if you are certain you know what you are doing.

B.1.3 Unnumbered Error Messages

Unnumbered messages appear when an error occurs that cannot be associated with a particular line of code. Generally these errors indicate problems with the command line, memory allocation, or file access. **MASM** may generate the unnumbered error messages below.

■ File-Access Errors

Any of the following errors may occur when **MASM** tries to access a file for processing. They usually indicate insufficient disk space, a corrupted file, or some other file error.

End of file encountered on input file

Include file *filename* not found

Read error on standard input

Unable to access input file: *filename*

Unable to open cref file: *filename*

Unable to open input file: *filename*

Unable to open listing file: *filename*

Unable to open object file: *filename*

```
Write error on cross-reference file

Write error on listing file

Write error on object file
```

■ Command-Line Errors

Any of the following errors may occur if you give an invalid command line when starting **MASM**.

```
Buffer size expected after B option

Error defining symbol "name" from command line

Extra file name ignored

Line invalid, start again

Path expected after I option

Unknown case option: option

Unknown option: option
```

■ Miscellaneous Errors

The following errors indicate a problem with memory allocation or some other assembler problem that is not related to a specific source line.

`Internal error`

Note the conditions when the error occurs and contact Microsoft Corporation by using the Product Assistance Request form at the end of this manual.

`Internal error - Problem with expression analyzer`

This problem may indicate an expression that **MASM** does not understand. Note the conditions when the error occurs and contact Microsoft Corporation by using the Product Assistance Request form at the end of this manual.

`Internal unknown error`

This error may indicate that the internal error table has been corrupted and **MASM** cannot figure out what the error is. Note the conditions when the error occurs and contact Microsoft Corporation by using the Product Assistance Request form at the end of this manual.

The following errors indicate a problem with memory allocation or some other assembler problem not related to a specific source line.

`Number of open conditionals: <number>`

Conditional-assembly directives (starting with **IF**) were given without corresponding **ENDIF** directives.

`Open procedures`

A **PROC** directive was given without a corresponding **ENDP** directive.

`Open segments`

A segment was defined, but never terminated with an **ENDS** directive. This error does not occur with simplified segment directives.

`Out of memory`

All available memory has been used, either because the source file is too long, or because there are too many symbols defined in the symbol table.

You can solve this problem in several ways. First, try assembling with no listing or cross-reference file. If this works, you can reassemble by specifying a null object file to get a listing or cross-reference file. You can also rewrite the source file to require less symbol space. Techniques for reducing symbol space include minimizing use of macros, equates, and structures; using short symbol names; using tab characters in macros rather than series of spaces; using macro comments (;;) rather than normal comments (;); and purging macro definitions after last use.

B.1.4 MASM Exit Codes

The assembler returns one of the following codes after an assembly. The codes can be tested by a make file or batch file.

Code	Meaning
0	No error
1	Argument error
2	Unable to open input file
3	Unable to open listing file
4	Unable to open object file
5	Unable to open cross-reference file

6	Unable to open include file
7	Assembly error
8	Memory-allocation error
10	Error defining symbol from command line (/D)
11	User interrupted

Note that if the exit code is 7, **MASM** automatically deletes the invalid object file.

B.2 CREF Error Messages and Exit Codes

The Microsoft Cross-Reference Utility, **CREF**, terminates operation and displays one of the following messages when it encounters an error:

`Can't open cross-reference file for reading`

The cross-reference file was not found. Make sure the file is on the specified disk and that the name is spelled correctly in the DOS command line.

`Can't open listing file for writing`

The disk is full or write protected, a file with the specified name already exists, or the specified device is not available.

`CREF has no options`

An option was specified in the command line with the slash (/) or dash (-) character, but **CREF** has no options.

`Extra file name ignored`

More than two files were specified on the command line. **CREF** uses only the first two files given.

`Line invalid, start again`

No cross-reference file was provided in the command line or at the prompt. **CREF** displays this message followed by a prompt asking for the file.

`Out of heap space`

CREF did not find enough memory to process the files. Try again with no resident programs or DOS shells, or add more memory.

`Premature EOF`

The file specified was not a valid cross-reference file, or the file was damaged.

`Read error on standard input`

A CONTROL-Z was received from the keyboard or from a redirected file.

CREF only returns two exit codes: 0 if the program is successful, or 1 if an error occurs.

INDEX

& (ampersand), operator, 226
<> (angle brackets), operator, 202, 216, 228
* (asterisk), operator, 175, 283
@ ("at sign"), 68
{ } (braces), xxvi
[ ] (brackets), xxvi
| (bar), xxvi
: (colon), operator
  definition, 181
  See also : (segment-override operator)
$ (dollar sign)
  location counter symbol, 137
  symbol names, used in, 68
... (dots), xxvi
= (equal sign), directive, 32, 162, 213
! (exclamation point), operator, 229
/ (forward slash), operator, 175
– (minus sign), operator, 175
% (percent sign)
  expression operator, 230
  symbol names, used in, 68
. (period), 68
+ (plus sign), operator, 175
? (question mark), 68
: (segment-override operator)
  definition, 181
  memory operands, with, 276, 279
  OFFSET operator, with, 185
  String instructions, with, 366
  XLAT instructions, with, 290
;; (semicolons), operator, 231
_ (underscore), 68

10-byte temporary-real format, 134
16-bit
  addressing modes, 282
  segments, 88, 98
.186 directive, 76
.286P directive, 76, 412
.287 directive, 73, 77, 132, 388
32-bit
  addressing modes, 269, 282
  segments, 88, 98, 260, 300
.386P directive, 76, 88, 98, 412
.387 directive, 73, 77, 132, 388
80186 processor described, 258
80286 processor described, 258
80287 processor described, 258
80386 processor
  32-bit
    addressing modes, 269, 282
    pointers, 129
    registers, 269
    segments, 88, 98, 260, 300
  .386 directive, 76, 98, 412
  bit-scan instructions, 324
  bit-test instructions, 323, 341
  BSF instruction, 324
  BSR instruction, 324
  BT instruction, 341
  BTC instruction, 341
  BTR instruction, 341
  BTS instruction, 341
  bytes, setting conditionally, 345
  CDQ instruction, 293
  CWDE instruction, 293
  data conversion, 293, 294
  described, 258
  DOS, using under, 269
  double shifts, 330
  enhanced instructions, 269
  IMUL instruction, 314
  LFS instruction, 297
  LGS instruction, 297
  loading pointers, 297
  LSS instruction, 297
  MOVSX instruction, 294
  MOVZX instruction, 294
  new instructions, 269
  PUSHAD and POPAD instructions, 303
  PUSHD and POPD instructions, 302
  registers, 261, 414
  scaling, 296
  SETcondition instruction, 345
  SHLD instruction, 330
  SHRD instruction, 330
  simplified segment directives, with, 88
  special registers, 414
80387 processor, described, 258
.8086 directive, 75
.8087 directive, 73, 77, 132, 388
8087 processor described, 258
8087/80287/80387 instruction set, 30
8087-family registers, 268
8088/8086 processors described, 257

/A option, 30, 96
AAA instruction, 317
AAD instruction, 318
AAM instruction, 317
AAS instruction, 317
ABS type, 161
Absolute segments, 101
Accumulator registers, 264
ADC instruction, 307, 309
ADD instruction, 307, 309
Adding, 307
Addition operator (+), 175
Addresses
  assembly listing, 43
  effective, 276, 279
Addressing modes
  16-bit, 282
  32-bit, 269
Adjusting masks, 329
Advisory warnings, 39
Aliases, 216
ALIGN directive, 138, 257
Align type, 98, 102
Alignment, of segments, 98, 138
.ALPHA directive, 96
Ampersand (&), operator, 226
AND instruction, 320, 321, 340
AND operator, 179
Angle brackets (<>), operator, 202, 216, 228
Arguments
  macros, 218, 219, 234
  passing on stack, 349
  repeat blocks, 223
Arithmetic operators, 175
Arrays
  boundary checking, 361
  defining of, 135
ASCII
  format for text files, 14
  name for unpacked BCD numbers, 316
Assembler. *See* MASM
Assembly listing
  false conditionals, 246
  macros, 247
  page breaks, 243
  page length, 243
  page width, 243
  Pass 1, 32
  reading, 42
  subtitle, 243
  suppressing, 245
  title, 242
ASSUME directive, 15, 107, 109, 181
Asterisk (*), operator, 175, 283
AT combine type, 101
"At" sign (@), 68
AUTOEXEC.BAT file, 10, 27, 28
Auxiliary-carry flag, 267
AX register, 264

/B option, 31
Backup copies, 7
Bar (|), xxvi
Base registers, 278, 282
Based operands, 278
Based-indexed operands, 278
BASIC compiler, 131
BASIC interpreter, 11, 131
BASIC language, mentioned, 334–354
BCD (binary coded decimal) numbers
  calculations with, 316, 394
  constants, 72
  coprocessor, with, 388
  defining of, 127
  variables initialized, 70
Binary coded decimals. *See* BCD
Binary files, 11
Binary radix, 71
Binary to decimal conversion, 318
BIOS (basic input/output system), xxvi
BIOS interrupts, 356
Bit fields, 143, 148
Bit mask, 319, 340
Bit-scan instructions, 324
Bit-test instructions, 341
Bits, rotating, 325
Bits, shifting, 325
Bitwise operators, 179
Bold type, xxv
Books, on assembly language, xxiii
Boolean bit operations, 320
BOUND instruction, 361
Boundary-checking array, 361
BP registers, 265
Braces ({ }), xxvi
Brackets ([ ]), xxvi
BSF instruction, 324
BSR instruction, 324
BT instruction, 341
BTC instruction, 341
BTR instruction, 341
BTS instruction, 341
Buffers
  defining, 135
  file, setting size, 31
Bugs, reporting, xxviii
BYTE align type, 98
BYTE type specifier, 119

C compiler, 131
C language, 84
C language, mentioned, 334–354
/C option, 35
Calculation operators, 174
CALL instruction, 122, 298, 347
Call tables, 347
Capital letter
  notation, xxv
  small, xxvii
  See also Case
Carry flag, 267, 308, 309, 311
Case
  case sensitivity, 42
  Case-sensitive compilers, 37
  Case-sensitivity options
    for LINK, 37
    for MASM, 37
  emulating Pascal statement, 334
CBW instruction, 292
CDQ instruction, 293
Character constant, 74
Character set, 68
Class type, 104
Classical-stack operands, coprocessor, 383
CLC instruction, 309, 311
CLD instruction, 365
CLI instruction, 358
CMP instruction, 335, 336, 345
CMPS instruction, 371
Code, assembly listing, 42
CODE class name, 86, 104
.CODE directive, 15, 89
@code equate, 91
Code equate, 91
Code segments
  defining, 89
  developing programs, in, 15, 16
  initializing, 111
  register, 263
  See also Segments
@codesize equate, 91
Codesize equate, 91
CodeView debugger
  code segments, 104
  development cycle, 13
  local variables, 351, 353
  segmented addresses, 260
  summary, 20
  symbolic information, 41
.COM format
  choosing, 11
  converting to, 19
  debugging, 42
  example, 16
.COM format *(continued)*
  initializing, 111
  segment types, effect of, 98
  tiny memory model, 84
Combine type, 100, 102
COMENT object record, 86, 169
COMM directive, 159, 166
Command lines
  with CREF, 53
  with MASM, 23
Command-line help, 34
COMMENT directive, 67
Comments, writing, 67
COMMON combine type, 100
Communal symbols, 159, 165
Compact memory model, 84, 87
Compare instructions, 401
Comparing register to zero, 322
Comparing strings, 371
Compatibility
  IBM languages, xxiii
  language compilers, 425
  other assemblers, 424
  upward, 257
Compilers, using with MASM, xix
  See also BASIC compiler, C compiler, etc
Conditional directives
  assembly directives, 40, 199, 220
  assembly passes, 201, 205
  error directives, 199, 220
  macro arguments, 202, 203, 207, 208
  nesting, 200
  operators, 226
  symbol definition, 201, 207
  value of true and false, 200, 206
Conditional-error directives, 204
Conditional-jump instructions, 335, 400
Configuration strategy, 7
.CONST directive, 89
Constants, 69, 273, 327
Control data, coprocessor, 392
CONTROL-BREAK, 23
CONTROL-C, 23
Conventions for manual, xxiv
Conversion, binary to decimal, 318
Converting data sizes, 292
Coprocessor
  8086 family, 258
  architecture, 379
  control data, 392
  directives, 75
  emulator, 33
  loading data, 389
  loading pi, 392

Coprocessor *(continued)*
no-wait instructions, 406
operands, 382
/R options, 30
registers, 268
See also 8087, 80386, etc.
Copying data, 289
CREF
command line, 53
cross-reference listing file, 53
described, 53
development cycle, in, 13
directive (.CREF), 249
error messages, 449
exit codes, 450
invoking, 54
prompts, 54
summary, 17
Cross-reference files
comparing with listing, 43
output, 24
specifying, 35
See also CREF
CS: override, 37
CS Register, 263
@ curseg equate, 90
Curseg equate, 90
Customer support, xxviii
CWD instruction, 292
CWDE instruction, 293
CX Register, 265

/D option, 32, 431
DAA instruction, 319
DAS instruction, 319
Data bus, 257
Data conversion, 292
.DATA directive, 15, 89
.DATA? directive, 89
@ data equate, 91
Data equate, 91
Data segments
defining, 89
developing programs, 15, 16
initializing, 15, 112
registers, 264
See also Segments
Data-definition directives, 123
@ datasize equate, 87, 91
Datasize equate, 87, 91
DB directive, 123, 124, 127
DD directive, 123
Debugging. *See* CodeView Debugger
DEC instruction, 309, 310
Decimal, packed BCD numbers, 316
Decimal radix, 71
Decrementing, 309
Defaults
radix, 71
segment names, 88, 93
segment registers, 109
simplified segment, 92
types, 195
Defining symbols from command line, 32
Destination string, 366
Development cycle, 11
Device drivers, 11, 270
Devices, 24
DF directive, 123, 126
DGROUP group name
COMM directive, with, 167
DOSSEG, with, 113
simplified segments, with, 86, 89, 92
Direction flag, 268, 365
Directives
.186, 76
.286, 76, 412
.286P, 76
.287, 73, 77, 132, 388
.386, 76, 88, 98, 412
.386P, 76
.387, 73, 77, 132, 388
.8086, 75
.8087, 73, 77, 132, 388
ALIGN, 138, 257
.ALPHA, 96
ASSUME, 15, 107, 109, 181
.CODE, 15, 89
COMM, 159, 166
COMMENT, 67
conditional. *See* Conditional directives
.CONST, 89
.CREF, 249
.DATA, 15, 89
.DATA?, 89
data definition, 123
DB, 123, 124, 127
DD, 123
defined, 66
DF, 123, 126
DOSSEG, 15, 85, 96
DQ, 124, 126, 130
DT, 124, 126, 130
DW, 123, 124, 128
ELSE, 200
END, 15, 79, 88, 111
ENDIF, 200
ENDM, 218, 223, 224, 225
ENDP, 121, 347, 359

Directives *(continued)*
ENDS, 95, 97, 143
EQU, 43, 162, 215, 216
equal sign (=), 32, 162, 213
.ERR, 205
.ERR1, 205
.ERR2, 205
.ERRB, 207
.ERRDEF, 207
.ERRDIF, 208
.ERRE, 206
.ERRIDN, 208
.ERRNB, 207
.ERRNDEF, 207
.ERRNZ, 206
EVEN, 138, 257
EXITM, 222, 223
EXTRN, 121, 159, 161
.FARDATA, 89
.FARDATA?, 89
full segment, 83
global, 159, 164
GROUP, 15, 83, 106, 181
IF, 40, 200
IF1, 201, 241
IF2, 201, 241
IFB, 202
IFDEF, 201
IFDIF, 203
IFE, 200
IFIDN, 203
IFNB, 202
IFNDEF, 201
INCLUDE, 217, 235, 237
INCLUDELIB, 169
instruction set, 75
IRP, 224
IRPC, 225
LABEL, 122, 136
.LALL, 220, 247
.LFCOND, 40, 246
.LIST, 245
LOCAL, 220, 223
MACRO, 218
.MODEL, 15, 75, 87, 162
.MSFLOAT, 75, 132
NAME, 165
ORG, 16, 111, 137
%OUT, 241
PAGE, 243
PROC, 92, 121, 346, 359
PUBLIC, 121, 122, 159, 160
PURGE, 237
.RADIX, 71
RECORD 148
REPT, 223

Directives *(continued)*
.SALL, 220, 247
SEGMENT, 95, 97, 181
.SEQ, 96
.SFCOND, 40, 246
simplified segment, 15, 83
.STACK, 15, 88
STRUC, 143
SUBTTL, 243
.TFCOND, 40, 246
TITLE, 165, 242
.XALL, 220, 247
.XCREF, 249
.XLIST, 245
Disk setup, 9
Displacement, 278
DIV instruction, 314
Divide overflow interrupt, 355
Dividing, 314
Dividing by constants, 327
Division operator (/), 175
Do
emulating C statement, 343
emulating FORTRAN statement, 343
Documentation feedback card, xxviii
Dollar sign ($)
location counter symbol, 137
symbol names, used in, 68
DOS
80386 under, 269
devices, 24
functions, 15, 356
interrupts, 356
Program Segment Prefix (PSP), 16
segment-order convention, 85
SET command, 27, 28
DOSSEG directive, 15, 85, 96
/DOSSEG linker option, 86
Dots (...), xxvi
Double shifts, with 80386 processor, 330
DQ directive, 124, 126, 130
DS registers, 264
/Dsymbol option, 32
DT directive, 124, 126, 130
DT Register, 265
Dummy parameters
macros, 218, 219, 234
repeat blocks, 223
Dummy segment definitions, 105
DUP operator, 135, 144, 145, 150
DW directive, 123, 124, 128
DWORD align type, 98
DWORD type specifier, 119
DX Registers, 265

/E option, 33, 132
Effective address, 276, 279
Ellipsis dots (...), xxvi
ELSE directive, 200
Emulator, coprocessor, 33
Encoded real numbers, 73, 132
Encoding of instructions, 273
END directive, 15, 79, 88, 111
ENDIF directive, 200
ENDM directive, 218, 223, 224, 225
ENDP directive, 121, 347, 359
ENDS directive, 95, 97, 143
ENTER instruction, 354
Environment variables
  INCLUDE, 8, 26, 236
  INIT, 8
  LIB, 8
  LINK, 8
  MASM, 8, 27
  PATH, 8
  TMP, 8
EQ operator, 180
EQU directive, 43, 162, 215, 216
Equal sign (=), directive, 32, 162, 213
Equates
  defined, 213
  nonredefinable, 214
  predefined, 90
  redefinable, 213
  string, 216
.ERR directive, 205
.ERR1 directive, 205
.ERR2 directive, 205
.ERRB directive, 207
.ERRDEF directive, 207
.ERRDIF directive, 208
.ERRE directive, 206
.ERRIDN directive, 208
.ERRNB directive, 207
.ERRNDEF directive, 207
.ERRNZ directive, 206
Error lines, displaying, 41
Error messages
  assembly listing, 43
  CREF, 451
  MASM, 432
ES registers, 264
ESC instruction, 412
EVEN directive, 138, 257
Exclamation point (!), operator, 229
.EXE format, 10, 14, 42
EXE2BIN
  development cycle, in, 13
  summary, 19
Exit codes
  CREF, 450
Exit codes *(continued)*
  MASM, 448
EXITM directive, 222, 223
Exponent, part of real-number constant, 73
Exponentiation, with 8087-family coprocessors, 404
Expression operator (%), 230
Expressions, defined, 173
External names, 36
External symbols, 161
Extra segment, 264
EXTRN directive, 121, 159, 161

F2XM1 instruction, 404
FABS instruction, 398
FADD instruction, 394
FADDP instruction, 394
False conditionals, listing, 40, 246
Far pointers, 128, 296
FAR type specifier, 120
.FARDATA directive, 89
.FARDATA? directive, 89
@fardata equate, 91
@fardata? equate, 91
Fardata equate, 91
Fardata? equate, 91
Fatal errors, 205
FBLD instruction, 390
FBSTP instruction, 390
FCHS instruction, 398
FCOM instruction, 401
FCOMP instruction, 402
FCOMPP instruction, 402
FCOS instruction, 405
FDIV instruction, 396
FDIVP instruction, 397
FDIVR instruction, 397
FDIVRP instruction, 397
FIADD instruction, 394
FICOM instruction, 402
FICOMP instruction, 402
FIDIV instruction, 397
FIDIVR instruction, 397
Fields
  assembler statements, 65
  bit, 143, 148
  records, 148, 151
  structures, 144, 146
FILD instruction, 390
@filename equate, 91
Filename equate, 91
Files
  AUTOEXEC.BAT, 10, 27, 28
  binary, 11

Files *(continued)*
buffer, 31
cross-reference, 24, 35
include, 26, 35, 168, 235
library, 13, 18
listing, 24, 35, 242
object, 13, 18
PACKING.LST, 7, 9
SETUP.BAT, 9
source. *See* Source files
specifications, 235
Filling strings, 373
FIMUL instruction, 396
FINIT instruction, 406
First-in-first-out (FIFO), 298
FIST instruction, 390
FISTP instruction, 390
FISUB instruction, 395
FISUBR instruction, 395
Flags
loading and storing, 292
register, 266
FLD instruction, 389
FLD1 instruction, 392
FLDCW instruction, 393
FLDL2E instruction, 392
FLDL2T instruction, 392
FLDLG2 instruction, 392
FLDLN2 instruction, 392
FLDPI instruction, 392
FLDZ instruction, 392
Floating-point format
compatibility, 424
Floating-point numbers, 30, 33
See also Real numbers
FMUL instruction, 396
FMULP instruction, 396
For, emulating high-level-language statement, 343
FORTRAN compiler, 131
FORTRAN language, mentioned, 343–354
Forward references
defined, 191
during a pass, 49
labels, 192
variables, 194
Forward slash (/), operator, 175
FPATAN instruction, 405
FPREM instruction, 398, 404
FPTAN instruction, 405
Fraction, 73
FRNDINT instruction, 398
FS registers, 264
FSCALE instruction, 398
FSIN instruction, 405
FSINCOS instruction, 405
FSQRT instruction, 398
FST instruction, 389
FSTCW instruction, 393
FSTP instruction, 389
FSTSW instruction, 393
FSUB instruction, 394
FSUBP instruction, 395
FSUBR instruction, 395
FSUBRP instruction, 395
FTST instruction, 402
Full segment directives, 83
Functions
C, 346
Pascal, 346
FWAIT instruction, 388
FWORD type specifier, 119
FXAM instruction, 404
FXCH instruction, 389
FXTRACT instruction, 398
FYL2X instruction, 404
FYL2XP1 instruction, 405

GE operator, 180
General-purpose registers, 264
Getting strings from ports, 375
Global directives
defined, 159
illustrated, 164
Global scope, 159
Global symbols, 160, 161
GROUP directive, 15, 83, 106, 181
Group-relative segments, 107
Groups
assembly listing, 46
defined, 106
illustrated, 107
size restriction, 107
See also DGROUP group name
GS Registers, 264
GT operator, 180

/H option, 34
Hard disk setup, 8
Hardware interrupts, 358
Help, 34
Hexadecimal radix, 71
HIGH operator, 184
High-level languages, memory model, 84, 87
High-level-language compilers, xix
HLT instruction, 412
Huge memory model, 85, 87

/I option, 35, 236
IBM languages, compatibility, xxiii
IDIV instruction, 314
IEEE format, 73, 131, 132, 388
IF directives, 40, 200
IF1 directive, 201, 241
IF2 directive, 201, 241
IFB directive, 202
IFDEF directive, 201
IFDIF directive, 203
IFE directive, 200
IFIDN directive, 203
IFNB directive, 202
IFNDEF directive, 201
Immediate operands, 273
Implied operands, 383
Impure code, checking for, 37
IMUL instruction, 312, 313, 314
IN instruction, 303
INC instruction, 307
INCLUDE directive, 217, 235, 237
INCLUDE environment variable, 8, 26, 236
Include files, 235
  assembly listings, 43
  communal variables, 168
  setting search paths, 26, 35
  using, 235
INCLUDELIB directive, 169
Incrementing, 307
Indeterminate operand, 136
Index checking, 361
Index operator, 177
Index registers, 278, 282
Indexed operands, 278
INIT environment variable, 8
Initializing
  data segments, 15
  segment registers, 111
  variables, 124
INS instruction, 375
Instruction-pointer register (IP), 266, 333
Instructions
  AAA, 317
  AAD, 318
  AAM, 317
  AAS, 317
  ADC, 307, 309
  ADD, 307, 309
  AND, 320, 321, 340
  bit scan, 324
  bit test, 323, 341
  BOUND, 361
  BSF, 324
  BSR, 324
Instructions *(continued)*
  BT, 341
  BTC, 341
  BTR, 341
  BTS, 341
  CALL, 122, 298, 347
  CBW, 292
  CDQ, 293
  CLC, 309, 311
  CLD, 365
  CLI, 358
  CMP, 335, 336, 345
  CMPS, 371
  compare, 401
  conditional jump, 333, 400
  CWD, 292
  CWDE, 293
  DAA, 319
  DAS, 319
  DEC, 309, 310
  defined, 66
  DIV, 314
  ENTER, 354
  ESC, 412
  F2XM1, 404
  FABS, 398
  FADD, 394
  FADDP, 394
  FBLD, 390
  FBSTP, 390
  FCHS, 398
  FCOM, 401
  FCOMP, 402
  FCOMPP, 402
  FCOS, 405
  FDIV, 396
  FDIVP, 397
  FDIVR, 397
  FDIVRP, 397
  FIADD, 394
  FICOM, 402
  FICOMP, 402
  FIDIV, 397
  FIDIVR, 397
  FILD, 390
  FIMUL, 396
  FINIT, 406
  FIST, 390
  FISTP, 390
  FISUB, 395
  FISUBR, 395
  FLD, 389
  FLD1, 392
  FLDCW, 393
  FLDL2E, 392
  FLDL2T, 392

Instructions *(continued)*
FLDLG2, 392
FLDLN2, 392
FLDPI, 392
FLDZ, 392
FMUL, 396
FMULP, 396
FPATAN, 405
FPREM, 398, 404
FPTAN, 405
FRNDINT, 398
FSCALE, 398
FSIN, 405
FSINCOS, 405
FSQRT, 398
FST, 389
FSTCW, 393
FSTP, 389
FSTSW, 393
FSUB, 394
FSUBP, 395
FSUBR, 395
FSUBRP, 395
FTST, 402
FWAIT, 388
FXAM, 404
FXCH, 389
FXTRACT, 398
FYL2X, 404
FYL2XP1, 405
HLT, 412
IDIV, 314
IMUL, 312, 313, 314
IN, 303
INC, 307
INS, 375
INT, 274, 298, 356, 359
INTO, 356, 358
IRET, 298, 357, 359
IRETD, 359
JC, 308, 311
Jcondition, 336, 338, 340, 357
JCXZ, 335, 344, 371, 372
JEXCZ, 343
JMP, 16, 109, 192, 333
LAHF, 292
LDS, 296
LEA, 295
LEAVE, 354
LES, 296, 371
LFS, 297
LGS, 297
LOCK, 412
LODS, 374
logical, 320
LOOP, 343

Instructions *(continued)*
LOOPE, 343
LOOPNE, 343
LOOPNZ, 343
LOOPZ, 343
LSS, 297
MOV, 109, 289, 413
MOVS, 368
MOVSX, 294
MOVZX, 294
MUL, 312
NEG, 309, 310
NOP, 192, 411
NOT, 323
OR, 320, 322
OUT, 303
OUTS, 375
POP, 109, 298
POPA, 302
POPAD, 303
POPD, 302
POPF, 301
POPFD, 302
program-flow, 333
protected mode, 413
PUSH, 109, 298
PUSHA, 302
PUSHAD, 303
PUSHD, 302
PUSHF, 301
PUSHFD, 302
RCL, 326
RCR, 326
REP, 367, 373, 376
REPE, 367, 371, 372
REPNE, 367, 371, 372
REPNZ, 367, 371, 372
REPZ, 367, 371, 372
RET, 121, 274, 298, 350
RETF, 348
RETN, 348
ROL, 326
ROR, 326
SAHF, 292
SAL, 326
SAR, 326
SBB, 309, 311
SCAS, 370
SETcondition, 345
SHL, 326
SHLD, 330
SHR, 326
SHRD, 330
STD, 365
STI, 358
STOS, 373

Instructions *(continued)*
SUB, 309, 310, 311, 337
TEST, 335, 340, 345
timing of, 273
WAIT, 387, 412
XCHG, 290
XLAT, 290
XOR, 320, 322
Instruction-set directives, 75
INT instruction, 274, 298, 356, 359
Integers, 70, 393
Integers, with coprocessor, 388
Interrupt-enable flag, 268, 357
Interrupts, 355
INTO instruction, 356, 358
I/O protection level flag, 268
IP Registers, 266
IRET instruction, 298, 357, 359
IRETD instruction, 359
IRP directive, 224
IRPC directive, 225
Italics, xxvi

JC instruction, 308, 311, 339
Jcondition instruction, 336, 338, 340, 357
JCXZ instruction, 335, 344, 371, 372
JEXCZ instruction, 343
JMP instruction, 16, 109, 192, 333
JO, 308, 339, 358
Jump tables, 334
Jumping conditionally, 335

Keystroke macros, 28

/L option, 35
LABEL directive, 122, 136
Labels
defined, 120
macros, in, 221
near code, 120
procedures, 121
LAHF instruction, 292
.LALL directive, 220, 247
Language compiler compatibility, 425
Large memory model, 85, 87
LDS instruction, 296
LE operator, 180
LEA instruction, 295
Learning assembly language, xxiii
LEAVE instruction, 354
LENGTH operator, 188
LES instruction, 296, 371
.LFCOND directive, 40, 246
LFS instruction, 297
LGS Instruction, 297
LIB
development cycle, in, 13
environment variable, 8
summary, 18
Library files, 13, 18
License, 7
Line number data, 42
Line numbers in MASM listings, 42
LINK
development cycle, in, 13
environment variable, 8
summary, 18
.LIST directive, 245
Listing
false conditionals, 246
files, 24, 35, 242
format
addresses, 43
code, 42
cross reference, 55
described, 42
EQU directive, 43
errors, 43
groups, 46
include files, 43
LOCK directive, 43
macro expansions, 43
macros, 45
Pass 1, reading, 49
records, 45
REP directive, 43
segment override, 43
segments, 46
structures, 45
symbols, 47
macros, 247
Pass 1,creating, 32
subtitles in, 243
suppressing output, 245
suppressing tables, 37
tables, suppressing, 37
Literal-character operator (!), 229
Literal-text operator (<>), 202, 216, 228
Loading constants to coprocessor, 391
Loading coprocessor data, 389
Loading pointers, 297
Loading values from strings, 374
LOCAL directive, 220, 223
Local scope, 159
Local symbols in macros, 220
Local variables, in procedures, 351
Location counter, 119, 137, 139, 190

Location counter symbol, 137
LOCK directive, assembly listing, 43
LOCK instruction, 412
LODS instruction, 374
Logarithms, 404
Logical bit operations, 320
Logical instructions, 320
Logical operators, 320
LOOP instruction, 343
Loop
  while equal, 343
  while not equal, 343
LOOPE instruction, 343
LOOPNE instruction, 343
LOOPNZ instruction, 343
LOOPZ instruction, 343
LOW operator, 184
LSS instruction, 297
LT operator, 180

Macro Assembler. *See* MASM
Macro comment operator (;;), 231
MACRO directive, 218
Macro expansions, assembly listings, 43
Macros
  argument testing, 203, 208
  arguments, 218, 219, 234
  assembly listing, 45
  calling, 219
  communal variables, 168
  compared to procedures, 218
  defined, 213, 217
  efficiency penalty, 213
  exiting early, 222
  expansions in listing, 247
  keystroke, 28
  local symbols, 220
  nested, 227, 232
  operators, 226
  parameters, 218, 219, 234
  recursive, 203, 231
  redefining, 234, 237
  removing from memory, 237
  text, 216
MAKE, development cycle, in, 13
MASK operator, 153
Masking bits, 320, 340
MASM
  command line, 23
  cross-reference file, 53
  described, 23
  development cycle, in, 13
  environment variable, 8, 26, 27
  error messages, 430
  exit codes, 448
MASM *(continued)*
  invoking, 23
  options. *See* Options
  prompts, 25
  summary, 17
Math coprocessors, 30, 258, 379
Medium memory model, 84, 87
Memory access, coordinating, 387
MEMORY combine type, 100
Memory models, 84
Memory operands, 276
Memory operands, coprocessor, 384
Memory requirements, xx
Messages
  error. *See* Error messages
  output, 28
  status, 429
  suppressing, 38
Messages to screen, 241
Microsoft Binary format, 131, 132
Microsoft Binary Real format, 73, 388
Minus operator (-), 175
Mixed-languages programs, 83, 96
/ML option, 36, 159
/ML option, MASM, 104
Mnemonics
  defined, 66
  reserved names, as, 69
MOD operator, 175
.MODEL directive, 15, 75, 87, 162
Modes, addressing. *See* Addressing modes
Modular programming, 159
Modulo division, 398
Modulo division operator, 175
MOV instruction, 109, 289, 413
Moving strings, 368
MOVS instruction, 368
MOVSX instruction, 294
MOVZX instruction, 294
MS-DOS, version requirements, xx
.MSFLOAT directive, 75, 132
/MU option, 36
MUL instruction, 312
Multiple modules, 164
Multiplication operator (*), 283
Multiplication operators, 175
Multiplying, 312
Multiplying by constants, 327
Multiword values, shifting, 329
/MX option, 36, 159
/MX option, MASM, 104

/N option, 37
NAME directive, 242

Names
  Assigning, 68
  external, 36
  public, 36
  reserved, 68, 235, 237
NE operator, 180
Near pointers, 128, 295
NEAR type specifier, 120
NEG instruction, 309, 310
Negating, 310
Nested-task flag, 268
Nesting
  conditionals, 200
  DUP operators, 135
  include files, 236
  macros, 227, 232
  procedures for Pascal, 354
  segments
    115
New features, 419
Nonredefinable equates, 214
NOP instruction, 192, 411
NOT instruction, 323
NOT operator, 179
Notational conventions, xxiv
NOTHING, ASSUME, 109
No-wait coprocessor instructions, 406
Null class type, 105
Null string, 220
Numbers. *See* Real numbers, signed numbers, etc.

Object files, 13, 18
Object records, 85, 169
Octal radix, 71
OFFSET operator, 92, 185
OFFSET operator, with group-relative segments, 107
ON GOSUB, emulating BASIC statement, 334
Opcode. *See* Instructions
Operands
  based, 278
  based indexed, 278
  based indexed with displacement, 278
  classical stack, 383
  coprocessor, 382, 383
  defined, 66, 173, 273
  immediate, 273
  implied, 383
  indeterminate, 136
  indexed, 278
  indirect memory, 273, 276, 278
  location counter, 190
Operands *(continued)*
  memory, 273, 276, 278
  record field, 154
  records, 151
  register, 261, 273, 274
  register indirect, 278
  relocatable, 276
  strong typing, 194
  structures, 146
  undefined, 136
Operators
  addition, 175
  AND, 179
  arithmetic, 175
  bitwise, 179
  calculation, 174
  defined, 173
  division (/), 175
  DUP, 135, 144, 145, 150
  EQ, 180
  expression (%), 230
  GE, 180
  GT, 180
  HIGH, 184
  index, 177
  LE, 180
  LENGTH, 188
  literal character (!), 229
  literal text (<>), 228
  logical, 320
  LOW, 184
  LT, 180
  macro comment (;;), 231
  MASK, 153
  minus (-), 175
  MOD, 175
  multiplication (*), 175
  NE, 180
  NOT, 179
  OFFSET, 92, 185
  OR, 179
  plus (+), 175
  precedence, 189
  PTR, 182, 193
  relational, 180
  SEG, 106, 167, 184
  segment override (:). *See* : (segment-override operator)
  shift, 178
  SHL, 178
  SHORT, 183, 192
  SHR, 178
  SIZE, 188
  structure-field name, 176
  substitute (&), 226
  subtraction, 175

Operators *(continued)*
THIS, 183
.TYPE, 186
TYPE, 187
WIDTH, 153
XOR, 179
Options
/A, 30, 96
/B, 31, 96
/C, 35
/D, 32, 431
/DOSSEG linker, 86
/Dsymbol, 32
/E, 33, 132
/H, 34
/I, 35, 236
/L, 35
/ML, 36, 159
/ML, MASM, 104
/MU, 36
/MX, 36, 159
/MX, MASM, 104
/N, 37
/P, 37
precedence, 27
/R, 30, 132
/S, 30, 96
summary, 29
/T, 38, 430
using, 23
/V, 38, 429
/W, 39, 195
/X, 40, 246
/Z, 41
/ZI, 41
OR instruction, 320, 322
OR operator, 179
ORG directive, 16, 111, 137
%OUT directive, 241
OUT instruction, 303
Output messages to screen, 241
OUTS instruction, 375
Overflow flag, 268, 308
Overflow interrupt, 355
Override, CS:, 37

/P option, 37
Packed BCD numbers, 127, 316, 319
See also BCD (binary coded decimal) numbers
Packed decimal integers, 70
Packed decimal numbers, 72
PACKING.LST file, 7, 9
PAGE align type, 98
Page breaks in assembly listings, 243
PAGE directive, 243
Page format of listing files, 242
PARA align type, 98
Parameters
defining in procedures, 349
macros, 218, 219, 234
repeat blocks, 223
Parity flag, 267
Partial remainder, 398
Pascal compiler, 131
Pascal language, mentioned, 334–354
Pass 1 listing, 32, 49
PATH environment variable, 8
PC-DOS. *See* DOS and MS-DOS
Percent sign (%)
expression operator, 230
symbol names, used in, 68
Period (.), 68
Phase errors, 32, 49
Pi, loading to coprocessor, 392
Placeholders, xxvi
Plus sign (+), operator, 175
Pointers
defining, 128
loading, 295
POP instruction, 109, 298
POPA instruction, 302
POPAD instruction, 303
POPD instruction, 302
POPF instruction, 301
POPFD instruction, 302
Ports
defined, 303
getting strings from, 375
sending strings to, 375
Precedence of operators, 189
Precedence, options, 27
Preserving case sensitivity, 36
PRIVATE combine type, 101
PROC directive, 92, 121, 347, 359
PROC type specifier, 120, 162
Procedures .
compared to macros, 218
defining labels, 121
Pascal, 346
using, 346
Processor directives, 75
Processors. *See* Coprocessors
Product Assistance Request form, xxviii
Program Segment Prefix (PSP), 16
Program-development cycle, 11
Program-flow instructions, 333
Prompts, with CREF, 54
Protected mode, 258, 259, 407
Protected-mode instructions, 413

Pseudo-op. *See* Directives
PTR operator, 182, 193
PUBLIC combine type, 100
PUBLIC directive, 121, 159, 160
Public names, 36
Public symbols, 160
PURGE directive, 237
PUSH instruction, 109, 298
PUSHA instruction, 302
PUSHAD instruction, 303
PUSHD instruction, 302
PUSHF instruction, 301
PUSHFD instruction, 302

Question mark (?), 68
QuickBASIC compiler, 131
Quotation marks (" "), xxvii
QWORD type specifier, 119

/R option, 30, 132
.RADIX directive, 71
Radixes
  binary, 71
  default, 71
  specifiers, 71
RCL instruction, 326
RCR instruction, 326
Real mode, 257, 259, 411
Real numbers
  arithmetic calculations, 393
  coprocessor, 388
  designator (R), 130
  encoding, 73, 132
  format, 30, 33, 73
  format, compatibility, 424
RECORD directive, 148
Record type, 148
Records
  assembly listing, 45
  declarations, 148
  defining, 143, 150
  field operands, 154
  fields, 151
  initializing, 148, 150, 151
  MASK operator, 153
  object, 85
  operands, 151
  variables, 150
  WIDTH operator, 153
Recursive macros, 203, 231
Redefinable equates, 213
Redefining interrupts, 358
Redefining macros, 234
Registers
Registers *(continued)*
  80386, 261
  80386, special, 414
  8087 family, 268
  accumulator, 264
  AX, 264
  base, 278, 282
  BP, 265
  BX, 265
  coprocessor, 268, 380, 381
  CS, 263
  CX, 265
  DI, 265
  DS, 264
  DX, 265
  ES, 264
  flags, 266
  FS, 264
  general purpose, 264
  GS, 264
  index, 278, 282
  IP, 266, 333
  mixing 16-bit and 32-bit, 283
  operands, 261, 273, 274
  operands, coprocessor, 385
  register-pop operands, coprocessor, 386
  reserved names, as, 69
  segment, 111, 263
  SI, 265
  SP, 266
  special, 414
  SS, 264
Relational operators, 180
Relocatable operands. *See* Memory operands
REP directive, assembly listing, 43
REP instruction, 367, 373, 376
REPE instruction, 367, 371, 372
Repeat blocks
  arguments, 223
  defined, 213, 223
  parameters, 223
  repeat for each argument, 224
  repeat for each character of string, 225
  repeat for specified count, 223
Repeat, emulating Pascal statement, 343
Repeat, using 8086-family string functions, 365
REPNE instruction, 367, 371, 372
REPNZ instruction, 367, 371, 372
Reporting problems, xxviii
REPT directive, 223
REPZ instruction, 367, 371, 372

Reserved names, 68, 235, 237
Resident programs, 270
Resume flag, 268
RET instruction, 121, 274, 298, 346
RETF instruction, 348
RETN instruction, 348
ROL instruction, 326
ROMable code, 11
ROR instruction, 326
Rotating bits, 325
Routines, FORTRAN, 346

/S option, 30, 96
SAHF instruction, 292
SAL instruction, 326
.SALL directive, 220, 247
SAR instruction, 326
SBB instruction, 309, 311
Scaling, 296
Scaling by powers of two, 398
Scaling factor, 283
SCAS instruction, 370
Search paths
  include files, 236
  setting, 26, 35
Searching strings, 370
Sections in assembly listings, 243, 244
SEG operator, 106, 167, 184
SEGMENT directive, 95, 97, 181
Segment-order method, 96
Segments
  16-bit, 88, 98
  32-bit, 88, 98, 260, 300
  absolute, 101
  alignment, 98, 138
  assembly listing, 46
  combine types, 100
  defined, 83
  definition, 95
  developing programs, 15, 16
  extra, 264
  group-relative offset, 107
  groups, 106
  initializing, 15
  MEMORY, 100
  nesting, 115
  ordering, 30, 104
  ordering, compatibility, 426
  override, assembly listings, 43
  override operator (:). *See*
    : (segment-override operator)
  registers, 263
  selectors, 260
  size, 98
  types, 98
Selectors, segment, 260
Semicolons (;;), operator, 231
Sending strings to ports, 375
.SEQ directive, 96
Serious warnings, 39
SET command (DOS), 27, 28
SET condition instruction, 345
Setting file buffer size, 31
Setting register to zero, 323
Setup, disk, 8, 9
SETUP.BAT file, 9
Severe errors, 39, 205
.SFCOND directive, 40, 246
Shift operators, 178
Shifting bits, 325
Shifting multiword values, 329
SHL instruction, 326
SHL operator, 178
SHLD instruction, 330
SHORT operator, 183, 192
SHR instruction, 326
SHR operator, 178
SHRD instruction, 330
SI registers, 265
Sign flag, 267, 311
Signed numbers, 124, 292, 308, 310
Sign-extending, 294
Simplified segment defaults, 92
Simplified segment directives, 15, 83
SIZE operator, 188
Small capitals, xxvii
Small memory model, 84, 87
Source files
  defined, 14
  format, 65
  illustrated, 14, 16
  include, 235
Source modules, 13, 159
Source string, 366
SP registers, 266
Special registers, 414
Square root, 398
SS registers, 264
STACK combine type, 100
Stack
  defined, 298
.STACK directive, 15, 88
Stack
  frame, 354
  operands, coprocessor, 383
  registers, 382
  segment, 15, 88, 100, 264
  segment, initializing, 114
  use of, 301
Standard output device, 28, 241, 431
Statement fields, 65

Statements, defined, 65
Statistics, 38, 429
Status messages, 429
STD instruction, 365
STI instruction, 358
Storing coprocessor data, 389
STOS instruction, 373
Strict type checking, 425
Strings
  comparing, 371
  constants, 74, 273
  defined, 365
  destination strings, 366
  equates, 216
  filling, 373
  getting from ports, 375
  loading values from, 374
  moving 368
  null, 220
  ports, transfer from and to, 375
  searching, 370
  source, 366
  structures, in, 144
  variables, 127
Strong typing, xix, 194
STRUC directive, 143
Structure type, 143
Structure-field-name operator, 176
Structures
  assembly listing, 45
  declarations, 143
  definitions, 143, 145
  fields, 146
  initializing, 143, 145, 146
  operands, 146
  overview, 143, 147
  variables, 145
SUB instruction, 309, 337
Subprograms, BASIC, 346
Subroutines, BASIC, 346
Substitute operator (&), 226
Subtitles in listings, 243
Subtracting values, 309
Subtraction operator, 175
SUBTTL Directive, 243
Summary
  CodeView, 20
  CREF, 17
  CV, 20
  EXE2BIN, 19
  LIB, 18
  LINK, 18
  MASM, 17
  options, 29
Switch, emulating C statement, 334
Symbol space, 429
Symbolic information, 41
Symbols
  assembly listing, 47
  communal, 159, 165
  defined, 67
  defining from command line, 32
  external, 161
  global, 160, 161
  location counter, 137
  public, 160
  relocatable operands, 276
SYMDEB, 41, 160
Syntax conventions, xxiv
System requirements, xx

/T option, 38, 430
TBYTE type specifier, 119
Temporary real format, 134
Terminate-and-stay-resident programs, 270
TEST instruction, 335, 340, 345
Testing bits, 341
Text editor, 13, 14, 28
Text equates. *See* String equates
Text Macros, 216
.TFCOND directive, 40, 246
THIS operator, 183
Timing of instructions, 273
Tiny memory model, 84
TITLE directive, 242
TMP environment variable, 8
Transcendental calculations, 404
Trap flag, 268, 357
Trigonometric functions, 404
Tutorial books, assembly language, xxiii
Two's complement, 124
Type
  ABS, 161
  align, 98, 102
  checking, strict, 425
  class, 104
  combine, 100, 101
  data, 42
  null class, 105
  operand matching, 194
.TYPE operator, 186
TYPE operator, 187
Type
  operators, 182
  PROC, 162
  record, 148
Type specifiers, 119
Type
  specifiers, 162

Type *(continued)*
  structure, 143
  use, 98
  USE, 282

Unary minus, 175
Unary plus, 175
Undefined operand, 136
Underscore (_), 68
Unpacked BCD numbers, 127, 316
  See also BCD numbers
Unsigned numbers, 124, 292, 308, 310
Updates, xxviii
Uppercase. *See* Case
Upward compatibility, 257
Use type, 98
USE type, 282

/V option, 38, 429
Variables
  communal, 165
  defined, 123
  external, 161
  floating point, 130
  initializing, 124
  integer, 124
  local, 351
  pointer, 128
  public, 160
  real number, 130
  record, 150
  string, 127
  structure, 145
Vertical bar (|), xxvi
Virtual 8086 Mode flag, 268

/W option, 39, 195
WAIT instruction, 387, 412
Warning levels, 39, 195
Weak typing in other assemblers, 195
While, emulating high-level-language
    statement, 343
WIDTH operator, 153
Width, structures, 148
WORD align type, 98
WORD type specifier, 119

/X option, 40, 246
.XALL directive, 220, 247
XCHG instruction, 290
.XCREF directive, 249
XENIX, 259
XENIX compatibility
  /L and /C, 36
  /MU, 36
  /P, 38
  pathnames, with / (forward slash),
      236
  /SI, 30
XLAT instruction, 290
.XLIST directive, 245
XOR instruction, 320, 322
XOR operator, 179

/Z option, 41
Zero flag, 267
Zero-extending, 294
/ZI option, 41